Undergraduate Dissertation

Deep Q-learning

A thesis submitted in fulfillment of the requirements
for the degree of Bachelor of Mathematics

Author:
Luis Carlos Mantilla Calderón

Supervisor:
Ph.D. Mauricio Jose Junca Pelaez

Department of Mathematics

Bogotá, Colombia
December 8, 2021

Contents

Introduction

Reinforcement learning (RL) is an area of machine learning that involves an agent which can be in a set of states S and can take some actions A per state [1]. The agent repeatedly interacts with an environment when taking some action and receives a reward R_t as is shown in Figure 1, therefore facing a sequential decision making (multi-stage) problem. To reduce the complexity of these problems, in practice, one assumes that the agent's state only depends on the observation. Mathematically this means that the transition probabilities satisfy the Markov property: $\mathbb{P}(r, s \mid S_t, A_t) = \mathbb{P}(r, s \mid \mathcal{H}_t, A_t)$. With this condition, dynamic programming techniques can be used for studying such tasks. Solving a problem of this type would correspond to obtain a policy π such that it gives an optimal distribution of actions to take to maximize the reward $G_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \dots$ won by the agent. Compared to other machine learning paradigms, reinforcement learning considers scenarios where the feedback for certain actions can be delayed since early decisions can influence far-in-time interactions with the environment. Real-world examples where RL can be applied are portfolio optimization [2], optimal control for robots [3], and quantum control for quantum computing [4].

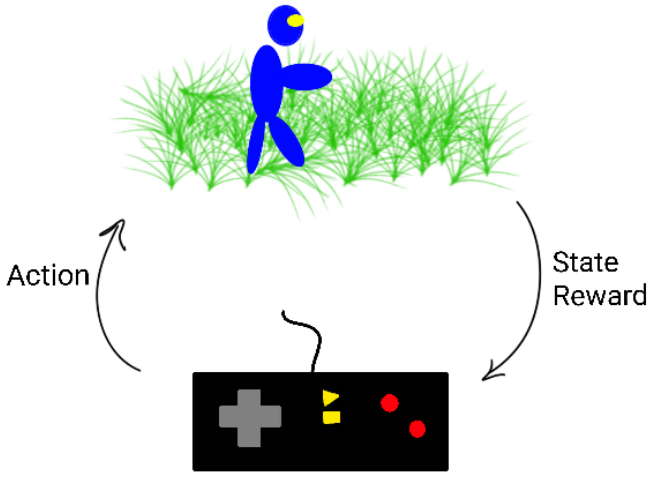


FIGURE 1: Interaction between environment (simulator) and agent (controller).

There are multiple ways of finding an optimal policy, one of them being Q-learning. It was first introduced by Watkins [5], and it is a reinforcement learning algorithm. Q-learning tackles the problem of finding the optimal actions that an agent should take on Markovian environments to maximize its reward without having an explicit model of the environment. The algorithm finds the "quality" function $Q : S \times A \to \mathbb{R}$ of every possible state-action pair (s, a) given by

$$Q_\pi(s,a) = \mathbb{E}_\pi[G_t \mid S_t = s, \quad A_t = a]. \tag{1}$$

The algorithm begins initializing a random Q function and then updates its values with the following rule:

$$Q\left(S_t, A_t\right) \leftarrow Q\left(S_t, A_t\right) + \alpha\left[R_{t+1} + \gamma \max_a Q\left(S_{t+1}, a\right) - Q\left(S_t, A_t\right)\right].$$

Here, $\alpha \in [0,1]$ is called the learning rate and controls how fast the Q-function changes per step, and $\gamma \in [0,1]$ is the discount factor and controls how quickly the agent takes rewarding steps. The discount factor can also be interpreted as preferring to get a reward r at a time t over getting the same reward r later in time. Eventually, this algorithm converges to a function that satisfies equation (1).

Some complex problems cannot be solved using classical Q-learning due to its enormous state space. For example, the game Go has approximately 10^{170} possible states. It would be impossible to learn such state-action functions employing Q-learning tables[1]. A way of overcoming this scalability problem is approximating the state-value functions with neural networks.

Neural networks (NNs) are a family of functions that consists of a combination of linear and non-linear functions capable of arbitrarily approximating any continuous function. The first result that shows this was Cybenko's universal approximation theorem [6], proving that the set generated by superpositions of linear and Sigmoid functions (σ):

$$\left\{ f(x) = \sum_{j=1}^{N} \alpha_j \sigma\left(y_j^{\mathrm{T}} x + \theta_j\right) \mid \alpha_j, \theta_j \in \mathbb{R}, y_j \in \mathbb{R}^n \right\}$$

is dense in $C([0,1]^N)$ with respect to the supremum norm. Nowadays, complex NN architectures are used for challenging tasks. For example, convolutional neural networks (CNNs) are used for image processing, recurrent neural networks (RNNs) for language processing, generative adversarial networks (GANs) for data augmentation, and more customized network architectures for particular tasks.

Mixing the ideas of function approximation using NNs, mainly, deep NNs (DNNs), with the field of reinforcement learning created the foundations for deep reinforcement learning. Neural networks allowed applying Q-learning in environments where classical Q-learning would have required an immense amount of resources. DeepMind made a step of applying such ideas to successfully train agents that could play multiple games with a vast state space, such as Atari games with Q-learning, and Chess (AlphaZero) and Go (AlphaGo) with other reinforcement learning techniques [7, 8, 9]. Efforts on mathematically understanding why the mixture of deep learning and Q-learning works have been made by Fan, et al. [10] and others, showing that the algorithm converges in certain simple cases. Ultimately, understanding in depth these tools will lead to further development of technology and provide welfare for our society.

[1] A Q-learning table refers to a table with a Q-value for every existing state-action pair.

Chapter 1

Reinforcement Learning

Reinforcement learning (RL) is an area of machine learning that studies how an agent learns to take optimal actions in an environment where multiple agent-environment interactions repeatedly occur. RL has shown a wide variety of applications, ranging from controlling a robotic arm to solving some tasks [11, 12]; to achieving super-human performance in numerous Atari games [7, 8]. This chapter introduces the basic concepts of reinforcement learning, such as a Markov decision process, the Bellman equation, dynamic programming, and finally, it will study temporal difference algorithms. We numerically implement an example using these algorithms with python. The reader can find the code on this link.

1.1 Markov decision processes

The core mathematical model behind reinforcement learning is a Markov decision process (MDP). An MDP contains a set of states $\mathcal{S}$, actions $\mathcal{A}$, and rewards R, and it describes the dynamics of the described system with a transition kernel $\mathcal{P}$ that dictates the time evolution of the states. Formally, we can define it as follows:

Definition 1.1.1. A **Markov decision process** is a tuple $(\mathcal{S}, \mathcal{A}, R, P)$ containing:

- A set of states $\mathcal{S}$ that can be discrete or continuous.
- A set of actions $\mathcal{A} = \bigcup_{s\in\mathcal{S}} A_s$, where A_{s_i} are the admissible actions for the state state $s_i \in \mathcal{S}$.
- A reward function $r : \mathcal{S} \times \mathcal{A} \longrightarrow \mathcal{D}[\mathbb{R}]$, where $\mathcal{D}[\mathbb{R}]$ denotes the set of probability distributions over $\mathbb{R}$.
- A transition kernel $P : \mathcal{S} \times \mathcal{A} \longrightarrow \mathcal{D}[\mathcal{S}]$, where $\mathcal{D}[\mathcal{S}]$ denotes the set of probability distributions over $\mathcal{S}$.

For an agent to evolve in an MDP, it must choose some action. This is defined with the concept of a policy:

Definition 1.1.2. A (stationary) **policy** $\pi : S \longrightarrow \mathcal{P}(\mathcal{A})$ is a function that outputs a probability density function over the action space $\mathcal{A}$ for a given $s \in \mathcal{S}$. We will denote Π the set of all possible policies.

In general, a policy could be time-dependent. Still, for stochastic MDPs with a bounded reward function, finite action and state space, and an objective function defined as the expected future reward, it is guaranteed that a stationary optimal[1] policy exist. Thus, for this thesis, we will only consider stationary policies. Depending

[1] We will explain what "optimal" means later in this section.

on the state space and the action space, there could be an enormous set of possible policies that an agent could follow. There are multiple ways to quantify how "good" a policy is over another one, and overall it depends on the primary goal of the MDP. For our purposes, we will consider the value function of a discounted MDP problem:

Definition 1.1.3. The **value function** $V^\pi : \mathcal{S} \longrightarrow \mathbb{R}$ is the expected total discounted reward a state $s \in \mathcal{S}$ obtains after following the policy π:

$$\begin{aligned} V^\pi(s) &= \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r(S_t) \;\middle|\; S_{t+1} \sim P(S_t, A_t), \quad A_t \sim \pi(S_t), \quad S_0 = s\right] \\ &= \mathbb{E}_s^\pi\left[\sum_{t=0}^{\infty} \gamma^t r(S_t)\right]. \end{aligned}$$

Here, $\gamma \in [0, 1]$ is known as the discount factor, and it controls how important is for an agent the immediate reward compared to a late reward. For values of γ near 1, a late reward will be important, while for values far from 1, late rewards become negligible.

A subtle but important generalization of the value function is the *state-action value function* or *quality function*. It quantifies how good a pair state-action is at a given time in some MDP:

Definition 1.1.4. The **state-action value function** $Q^\pi : \mathcal{S} \times \mathcal{A} \longrightarrow \mathbb{R}$ is the expected total discounted reward a state $s \in \mathcal{S}$ obtains after taking some action $a \in \mathcal{A}$, and then following the policy π:

$$\begin{aligned} Q^\pi(s, a) &= \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r(S_t) \;\middle|\; S_{t+1} \sim P(S_t, A_t), \quad A_t \sim \pi(S_t), \quad S_0 = s, \quad A_0 = a\right] \\ &= \mathbb{E}_{(s,a)}^\pi\left[\sum_{t=0}^{\infty} \gamma^t r(S_t)\right]. \end{aligned}$$

Both the value and the state-action functions are calculated with an infinite series (if no final state is ever reached) which is obtained by a complex chain of events in the MDP. However, we can break down these infinite-series calculations into a simpler recursive relation that both functions must satisfy, by simply evaluating a policy π on the value and the state-action functions:

$$\begin{aligned} V^\pi(s) &= \sum_{a \in A_s} \pi(a \mid s)\left\{r(s, a) + \gamma \sum_{s'} P(s' \mid s, a) V^\pi(s')\right\} \\ Q^\pi(s, a) &= r(s, a) + \gamma \sum_{s'} P(s' \mid s, a) V^\pi(s') \end{aligned}$$

respectively.

As we said previously, depending on the problem, there could be an immense set of possible policies, and if we want to solve an MDP problem, we must find an optimal policy $\pi^\star$ such that it maximizes the value function

$$V^\star(s) = \sup_{\pi \in \Pi} V^\pi(s), \quad \forall s \in \mathcal{S}$$

and consequently, maximizes the state-action function

$$Q^\star(s,a) = \sup_{\pi\in\Pi} Q^\pi(s,a), \quad \forall (s,a) \in \mathcal{S}\times\mathcal{A}.$$

These policies are obtained as the arguments of these optimization problems. By replacing the policy evaluation on the right-hand side we obtain what are known as the **Bellman equations of optimality**:

$$V^\star(s) = \sup_{a\in\mathcal{A}} \left\{ r(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V(s') \right\},$$
$$Q^\star(s,a) = \sup_{b\in\mathcal{A}} \left\{ r(s,a) + \gamma \sum_{s'} P(s' \mid s,a) Q(s',b) \right\}.$$

There are multiple techniques to solve these equations, and consequently, find the optimal policy $\pi^\star$, some of which we will study along this thesis. The next subsection will discuss dynamic programming, a technique that uses bootstrapping to solve this problem.

1.1.1 Dynamic programming

Two main algorithms fall in the category of dynamic programming: value iteration and policy iteration. Let us begin with the former; Value iteration is a method that, using the information of the transition kernel, performs bootstrapping on the Bellman equation of optimality for the value function to obtain an estimate of the real optimal value function of all states. With such a function, it is easy to obtain the optimal policy since the optimal action that an agent on a particular state must take is the one that maximizes the value function on the current state. Algorithmically, this can be written as:

Algorithm 1 Value Iteration

Require: $|r| < \infty$, $|\mathcal{S}| < \infty$, $|\mathcal{A}| < \infty$, $\epsilon \geq 0$.
Ensure: $V(s) \approx r(s,\pi^\star(s)) + \gamma \sum_{s'} P(s' \mid s, \pi^\star(s)) V(s')$
1: $V(s) \leftarrow rand(0,1)$ for all $s \in \mathcal{S}$.
2: $D \leftarrow$ **false**
3: **while** $D =$ **false do**
4: **for** $s \in \mathcal{S}$ **do**
5: $w_s \leftarrow V(s)$
6: $V(s) \leftarrow \max_a \{ r(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V(s') \}$
7: **end for**
8: **if** $\max_s |w_s - V(s)| \leq \epsilon$ **then**
9: $D \leftarrow$ **true**
10: **end if**
11: **end while**

This pseudocode can be implemented more efficiently by using a matrix representation for V, R, and P. Furthermore, the optimal policy[2] is obtained by setting

$$\pi^\star(s) = \text{argmax}_a \left\{ r(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V(s') \right\}.$$

Theorem 1.1.1. *The value function V_i obtained from the value iteration algorithm converges to the optimal value function $V^\star$ when $i \to \infty$.*

Proof. Define the operator $\mathcal{B} : \mathcal{S}^* \longrightarrow \mathcal{S}^*$ as

$$\mathcal{B}[V] := \max_{\mathbf{a} \in \mathcal{A}} \{ R_{\mathbf{a}} + \gamma \mathcal{P}_{\mathbf{a}} V \}$$

where $\mathbf{a} = (a_1, \ldots, a_n) \in \mathcal{A}$ is a set of actions for all state $(s_1, \ldots, s_n)$, $R_{\mathbf{a}}$ is the vector containing all rewards for every state s_i with action a_i, and $\mathcal{P}_{\mathbf{a}}$ is the matrix representation of the transition kernel for the set of actions $\mathbf{a}$. Let us check that $\mathcal{B}$ is a contraction under the norm $\|\cdot\|_\infty$:

Let $V_1, V_2 \in \mathcal{S}^*$ such that $\tilde{\mathbf{a}} = \text{argmax}_{\mathbf{a} \in \mathcal{A}} \{ R_{\mathbf{a}} + \gamma \mathcal{P}_{\mathbf{a}} V_1 \}$, then

$$\begin{aligned}
\|\mathcal{B}[V_1] - \mathcal{B}[V_2]\|_\infty &= \left\| \max_{\mathbf{a} \in \mathcal{A}} \{ R_{\mathbf{a}} + \gamma \mathcal{P}_{\mathbf{a}} V_1 \} - \max_{\mathbf{b} \in \mathcal{A}} \{ R_{\mathbf{b}} + \gamma \mathcal{P}_{\mathbf{b}} V_2 \} \right\|_\infty \\
&\leq \| R_{\tilde{\mathbf{a}}} + \gamma \mathcal{P}_{\tilde{\mathbf{a}}} V_1 - R_{\tilde{\mathbf{a}}} - \gamma \mathcal{P}_{\tilde{\mathbf{a}}} V_2 \|_\infty \\
&= \gamma \| \mathcal{P}_{\tilde{\mathbf{a}}} \| \| V_1 - V_2 \|_\infty \\
&= \gamma \| V_1 - V_2 \|_\infty < \| V_1 - V_2 \|_\infty.
\end{aligned}$$

Since $\mathcal{B}$ is a contraction, by Banach's fixed point theorem, there is a unique fixed point obtained by iteratively applying $\mathcal{B}$: $V^* = \lim_{m \to \infty} \mathcal{B}^m[V]$ for any $V \in \mathcal{S}^*$. In this case, the fixed point of $\mathcal{B}$ is the optimal value function $V^\star$ since it satisfies the Bellman equation:

$$\mathcal{B}[V^\star] = \max_{\mathbf{a} \in \mathcal{A}} \{ R_{\mathbf{a}} + \gamma \mathcal{P}_{\mathbf{a}} V^\star \} = V^\star.$$

□

A consequence from this theorem is that the policy π obtained from the value iteration algorithm converges almost surely to the optimal policy $\pi^\star$. With this in hand, we can quantify the error between the optimal value function and the value function obtained using Algorithm 1. The algorithm stops when $\left\| \mathcal{B}^n V - \mathcal{B}^{n+1} V \right\|_\infty \leq \epsilon$, this means that

$$\begin{aligned}
\| \mathcal{B}^n V - V^* \|_\infty &\leq \sum_{m=n}^{\infty} \left\| \mathcal{B}^m V - \mathcal{B}^{m+1} V \right\|_\infty \\
&\leq \sum_{m=n}^{\infty} \gamma^{m-n} \left\| \mathcal{B}^n V - \mathcal{B}^{n+1} V \right\|_\infty \\
&= \frac{\left\| \mathcal{B}^n V - \mathcal{B}^{n+1} V \right\|_\infty}{1-\gamma} \\
&\leq \frac{\epsilon}{1-\gamma}.
\end{aligned}$$

[2] We need to prove that it is optimal.

Another important dynamic programming algorithm is the policy iteration algorithm. It is a variation of the value iteration algorithm where the policy is directly optimized rather than induced from the value function.

Algorithm 2 Policy Iteration

Require: $|r| < \infty$, $|\mathcal{S}| < \infty$, $|\mathcal{A}| < \infty$, $\epsilon \geq 0$.
Ensure: $\pi(s) \approx \text{argmax}_a \{r(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V^\star(s')\}$
$V(s) \leftarrow rand(0,1)$ for all $s \in \mathcal{S}$.
$\pi(s) \leftarrow \mu(s)$ where $\mu \in \Pi$ is a random policy.
$D1 \leftarrow$ **false**
while $D1 =$ **false do**
  $D2 \leftarrow$ **false**
  while $D2 =$ **false do**
    for $s \in \mathcal{S}$ **do**
      $w_s \leftarrow V(s)$
      $V(s) \leftarrow \{r(s,\pi(s)) + \gamma \sum_{s'} P(s' \mid s,\pi(s)) V(s')\}$
    end for
    if $\max_s |w_s - V(s)| \leq \epsilon$ **then**
      $D2 \leftarrow$ **true**
    end if
  end while
  for $s \in \mathcal{S}$ **do**
    $\mu_s \leftarrow \pi(s)$
    $\pi(s) \leftarrow \text{argmax}_a \{r(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V(s')\}$
  end for
  if $\mu = \pi$ **then**
    $D1 \leftarrow$ **true**
  end if
end while

The performance of these two algorithms depends on the problem being solved, sometimes value iteration will outperform policy iteration in terms of convergence speed and vice-versa. Additionally, the convergence of policy iteration can be proven with similar arguments to the ones used for value iteration. We will now use dynamic programming to solve a simple MDP on a two-dimensional grid world.

1.1.2 Stochastic agent in 2D grid-world pt.1

To illustrate the value iteration algorithm, let us consider the following MDP in a 2D grid word. Suppose there is a drone you want to drive to a specific region A while avoiding a set of obstacles B. Additionally, this drone is subject to a stochastic wind biased in some direction. The MDP that allow us to model the drone and its environment consist of:

- States: the integer points inside a bounded 2D grid, in this case, a 15×15 grid.
- Actions: The drone can move up, down, left and right.
- Rewards: 100 points for states in region A, -50 points for states in region B and 0 elsewhere.
- Transition kernel:

$$P\left((z,w) \mid ((x,y),u)\right) = \begin{cases} 0.3 & \text{if } z = x, w = y+1 \\ 0.4 & \text{if } z = x, w = y+2 \\ 0.2 & \text{if } z = x-1, w = y+2 \\ 0.1 & \text{if } z = x-1, w = y+1 \end{cases}$$

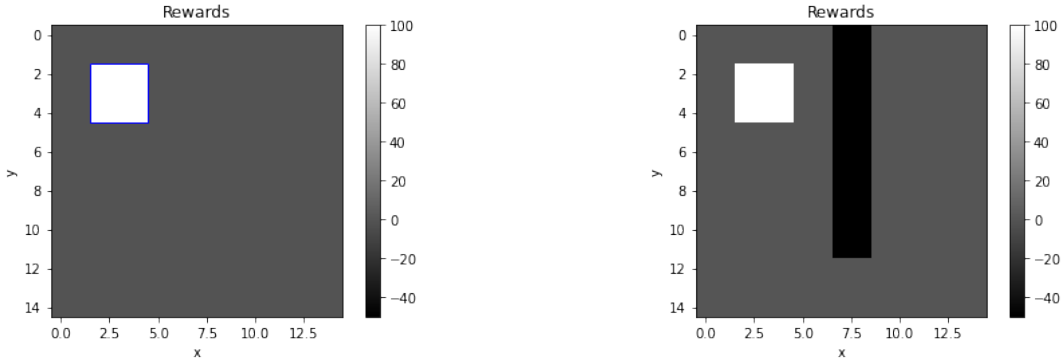
$$P\left((z,w) \mid ((x,y),d)\right) = \begin{cases} 0.3 & \text{if } z = x, w = y \\ 0.3 & \text{if } z = x, w = y-1 \\ 0.2 & \text{if } z = x-1, w = y \\ 0.2 & \text{if } z = x-1, w = y-1 \end{cases}$$

$$P\left((z,w) \mid ((x,y),l)\right) = \begin{cases} 0.3 & \text{if } z = x-1, w = y+1 \\ 0.2 & \text{if } z = x-1, w = y \\ 0.3 & \text{if } z = x-2, w = y \\ 0.2 & \text{if } z = x-2, w = y+1 \end{cases}$$

$$P\left((z,w) \mid ((x,y),r)\right) = \begin{cases} 0.3 & \text{if } z = x+1, w = y \\ 0.4 & \text{if } z = x+1, w = y+1 \\ 0.2 & \text{if } z = x, w = y \\ 0.1 & \text{if } z = x, w = y+1, \end{cases}$$

with the boundary condition of setting the drone in the same place if an invalid action is chosen. This transition kernel acts effectively as a wind pushing the drone to the upper right corner of the map.

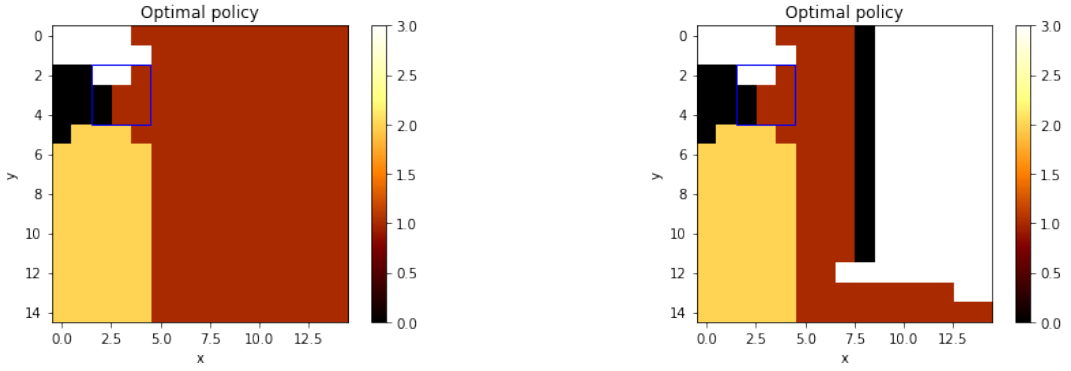
We studied two particular scenarios: a map without obstacles and a map with a simple rectangular obstacle, as shown in Figure 1.1. After performing value iteration with $\epsilon = 10^{-6}$ and $\gamma = 0.9$ we obtained the policies shown on 1.2. We can see that, on both scenarios, the left half of the map coincide, while the right half changes due to the barrier. On the former map, the drone goes a quickly as possible to the left half, while on the latter map, the drone surrounds the barrier (region B) before moving to region A.



(A) (B)

FIGURE 1.1: Rewards of two 2D grid-world instances. Figure (A) is an obstacle-free map and Figure (B) is a map with a rectangular obstacle obstructing the middle.

One of the main restrictions of using either the value iteration or policy iteration algorithm in real-life MDP problems is the lack of knowledge of the transition kernel. A naive way to avoid this problem would be to estimate the transition kernel by observing an agent interact with the environment, but this task is often hard to do.



(A) (B)

FIGURE 1.2: Optimal policy of the 2D grid-world instances shown in Figures 1.1a and 1.1b, respectively. Each color represents an optimal action: red refers to moving left, black to moving right, yellow and white to move up and down, respectively.

Accordingly, we will discuss in the next section a subset of model-free[3] algorithms known as *temporal-difference (TD) algorithms*.

1.2 Temporal-difference learning

In real-life problems that can be modeled with an MDP setup, obtaining the explicit transition kernel is often challenging; this is why model-free algorithms are successful and practical in a wide range of scientific fields. For solving reinforcement learning problems, the two main model-free algorithms are Monte-Carlo methods and temporal-difference methods. For example, to estimate a state's value function under some policy π, the Monte-Carlo approach would be to sample the obtained reward for multiple episodes and then average to estimate the value function

$$V^{\pi}(s) \leftarrow \frac{1}{N}\left(G_1 + G_2 + \cdots + G_N\right),$$

where N is the number of episodes and $G = R_0 + \gamma R_1 + \gamma^2 R_2 + \cdots \gamma^m R_m$ is the total discounted reward obtained until reaching a terminal state[4]. Similarly, suppose we wanted to estimate the value function of a state s using Monte-Carlo while forgetting old samples (maybe the MDP changes over time). In that case, we can update the value function every episode by a small amount

$$V^{\pi}(s) \leftarrow (1-\alpha)V^{\pi}(s) + \alpha(\underbrace{G_i}_{\text{sampled reward}}),$$

where $\alpha \in [0, 1]$ is the learning rate. However, one big problem that this approach has is the need to complete full episodes to update the value function of a single state. This issue becomes intractable when the trajectories of the agent are too long or even when the problem is non-episodic, meaning that there could be no terminal state. Here is where temporal-difference (TD) algorithms come into play.

TD learning uses the idea of bootstrapping (estimating a value out of some other estimate) to perform the task that Monte-Carlo RL achieves. For estimating V under

[3] Algorithms that do not require prior knowledge of the transition kernel.
[4] Monte-Carlo methods must reach a terminal state in order to do estimates.

some policy π, TD learning uses the next state's value function $V(s')$ to update the value function of a particular state $V(s)$:

$$V^{\pi}(s) \leftarrow (1-\alpha)V^{\pi}(s) + \alpha(\underbrace{R + \gamma V^{\pi}\left(s'\right)}_{\text{estimated reward}}).$$

Usually, $R+\gamma V\left(s'\right)$ is called the *TD target*, and in this example, the algorithm pushes the value function to get closer to the TD target value by some amount α. Another crucial algorithm that fits under the category of TD learning, and will be the main character of this thesis, is *Q-learning*. This model-free algorithm allows us to estimate value functions and optimize policies to maximize rewards on MDP problems.

1.2.1 Q-learning

We already mentioned how to evaluate a policy in a model-free manner for a given MDP using the value function. Furthermore, for optimizing a policy π in a model-free way, one must use the quality function Q, since the update

$$\pi(s) = \text{argmax}_a \left\{ r(s,a) + \sum_{s'} P(s' \mid s,a)V(s') \right\},$$

as used in the value iteration algorithm, uses knowledge of the MDP transition matrix. The way we overcome this is by using Q to choose the best action

$$\pi(s) = \text{argmax}_a \, Q(s,a).$$

Using both of these model-free methods of evaluating a policy and optimizing a policy give rise to the Q-learning algorithm first proposed by Watkins [5], whose goal is to achieve the optimal quality function $Q^*(s,a)$.

Algorithm 3 Q-learning with a given non-greedy policy π

Require: $|r| < \infty$, $|\mathcal{S}| < \infty$, $|\mathcal{A}| < \infty$, $\alpha \in (0,1]$, $\gamma \in (0,1]$, $N \geq 0$.
Ensure: $Q(s,a) \approx \max_{\pi} \mathbb{E}\left[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \ldots \mid s_t = s, a_t = a, \pi\right]$
1: $Q(s,a) \leftarrow rand(0,1)$ for all $(s,a) \in \mathcal{S} \times \mathcal{A}$.
2: $s \leftarrow rand(\mathcal{S})$: pick an initial state.
3: $i \leftarrow 0$
4: **while** $i \leq N$ **do**
5: Pick some admissible action a using a specified policy π.
6: Let s' and r be the new state and reward, respectively, after applying a.
7: $Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha\left(r + \gamma \max_b Q(s',b)\right)$
8: **if** s' is terminal **then**
9: $s \leftarrow rand(\mathcal{S})$
10: **else**
11: $s \leftarrow s'$
12: **end if**
13: **end while**

To proof the convergence of this algorithm, we will use the following theorem on stochastic processes:

Theorem 1.2.1 (Jaakkola, et al. [13])**.** *A stochastic iterative process*

$$\Delta_{i+1}(s) = (1-\alpha_i(s))\Delta_i(s) + \beta_i(s)F_i(s)$$

with state space $\mathcal{S}$ and random variable $F_i(s)$, converges to 0 when $i \to \infty$ with probability of 1 if the following statements hold:

- *The state space $\mathcal{S}$ is a finite set: $|\mathcal{S}| < \infty$*
- $\sum_i \alpha_i(s) = \infty$, $\sum_i \alpha_i^2(s) < \infty$, $\sum_i \beta_i(s) = \infty$, $\sum_i \beta_i^2(s) < \infty$.
- $\mathbb{E}[\beta_i(s) \mid \mathcal{Z}_i] \leq \mathbb{E}[\alpha_i(s) \mid \mathcal{Z}_i]$.
- $\|\mathbb{E}[F_i(s) \mid \mathcal{Z}_i]\|_W \leq \gamma \|\Delta_i\|_W$, *for* $0 < \gamma < 1$.
- $\mathrm{Var}[F_i(s) \mid \mathcal{Z}_i]] \leq c(1 + \|\Delta_i\|_W)^2$, *for* $c \in \mathbb{R}$.

where $\mathcal{Z}_i$ refers to all the history of the previous step (including previous values of β_i, α_i, and δ_i) and $\|\cdot\|_W$ is a weighted maximum norm $\|\cdot\|_W = \max_s \left(\frac{\cdot}{W(s)}\right)$.

Using this theorem, we can now proof the convergence of Q-learning.

Theorem 1.2.2. *Suppose that the learning rate satisfies*

$$\sum_i \alpha_i(s,a) = \infty \text{ and } \sum_i \alpha_i^2(s,a) < \infty$$

for all $s \in \mathcal{S}$ and $a \in A_s$, all the rewards are bounded $r(s,a) \leq R_{\max}$, and $|\mathcal{S}| < \infty$, then, when $N \to \infty$, the quality function obtained from the Q-learning algorithm converges to $Q^\star$ with a probability of 1.

Proof. Starting from the update of Q-learning

$$Q_{i+1}(s,a) = (1 - \alpha_i(s,a))Q_i(s,a) + \alpha_i(s,a)(r + \gamma \max_b Q_i(s',b))$$

subtract $Q^\star(s,a)$ from both sides and let $\Delta_i(s,a) = Q_i(s,a) - Q^\star(s,a)$ and $F_i(s,a) = r(s,a) + \gamma \max_b Q_i(s',b) - Q^\star(s,a)$, thus

$$\Delta_{i+1}(s,a) = (1 - \alpha_i(s,a))\Delta_i(s,a) + \alpha_i(s,a)F_i(s,a)$$

Comparing this equation with Theorem 1.2.1 and setting $\beta_i = \alpha_i$, we need to proof two inequalities to have a guarantee for convergence:

- $\|\mathbb{E}[F_i(s) \mid \mathcal{Z}_i]\|_W \leq \gamma \|\Delta_i\|_W$:
  By choosing W to be the uniform weighting we have the following:

$$\begin{aligned}
\|\mathbb{E}[F_i(s,a) \mid \mathcal{Z}_i]\|_\infty &= \left\| \sum_{s'} \mathbb{P}(s' \mid s,a) \left(r(s,a) + \gamma \max_b Q_i(s',b) - Q^\star(s,a) \right) \right\|_\infty \\
&= \left\| \sum_{s'} \mathbb{P}(s' \mid s,a) \left(r(s,a) + \gamma \max_b Q_i(s',b) \right) - Q^\star(s,a) \right\|_\infty
\end{aligned}$$

  using the fact that $Q^\star(s,a) = \sum_{s'} \mathbb{P}(s' \mid s,a) \left(r(s,a) + \gamma \max_b Q^\star(s',b)\right)$ we get that

$$\begin{aligned}
\|\mathbb{E}[F_i(s,a) \mid \mathcal{Z}_i]\|_\infty &= \left\| \sum_{s'} \mathbb{P}(s' \mid s,a) \left(\gamma \max_a Q_i(s',a) - \gamma \max_b Q^\star(s',b) \right) \right\|_\infty \\
&\leq \gamma \max_{(s,a)} \sum_{s'} \mathbb{P}(s' \mid s,a) \max_{(s',b)} \left| Q_i(s',b) - Q^\star(s',b) \right| \\
&= \gamma \|Q_i - Q^\star\|_\infty = \gamma \|\Delta_i\|_\infty.
\end{aligned}$$

- $\text{Var}[F_i(s) \mid \mathcal{Z}_i]] \leq c(1 + \|\Delta_i\|_W)^2$:

$$\begin{aligned}\text{Var}[F_i(s,a) \mid \mathcal{Z}_i]] &= \text{Var}[r(s,a) + \gamma \max_b Q_i(s',b) - Q^\star(s,a) \mid \mathcal{Z}_i]] \\ &= \text{Var}[r(s,a) + \gamma \max_b Q_i(s',b) \mid \mathcal{Z}_i].\end{aligned}$$

  Since the rewards are bounded and the state space if finite, the random variable $r(s,a)+\gamma \max_b Q_i(s',b)$ has an upper and lower bound U and L, respectively. We can use *Popoviciu's inequality* which provides an upper bound on the variance of a bounded probability distribution with upper and lower bounds U and L, respectively. Such bound is given by

$$\sigma^2 \leq \frac{1}{4}(U-L)^2,$$

  and we can use it to bound $\text{Var}[F_i(s,a) \mid \mathcal{Z}_i]]$. From this, it follows that for some $c \in \mathbb{R}$

$$\text{Var}[F_i(s,a) \mid \mathcal{Z}_i]] \leq c(1 + \|\Delta_i\|_W)^2.$$

This means that with probability 1, when $i \to \infty$, we have $\Delta_i \to 0$, which is equivalent to $Q_i \to Q^\star$. □

In practice, we cannot fulfill the requirements since one cannot visit infinitely many times all states. Still, this algorithm tends to work in reasonable-sized MDPs. Something to note about Q-learning is the update rule; it does not use the same *behavior policy* π used for exploring, but instead uses the greedy policy as the *target policy*. This type of learning is known as *offline learning* since both the behavior and target policies are different. A variation of the Q-learning algorithm that uses the same policy for both purposes is SARSA (Algorithm 4). This algorithm does *online learning*, meaning that the behavior policy is the policy that the agent learns.

Algorithm 4 SARSA with policy π

Require: $|R| < \infty$, $|\mathcal{S}| < \infty$, $|\mathcal{A}| < \infty$, $\alpha \in (0,1]$, $\gamma \in (0,1]$, $N \geq 0$.
Ensure: $Q(s,a) \approx \max_\pi \mathbb{E}\left[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \ldots \mid s_t = s, a_t = a, \pi\right]$

```
Q(s,a) ← rand(0,1) for all (s,a) ∈ S × A.
s ← rand(S) : pick an initial state.
i ← 0
while i ≤ N do
    Pick some admissible action a using π(s).
    Let s' and r be the new state and reward, respectively, after applying a.
    Pick some admissible action b using π(s').
    Q(s,a) ← (1 − α)Q(s,a) + α (r + γQ(s',b))
    if s' is terminal then
        s ← rand(S)
    else
        s ← s'
    end if
end while
```

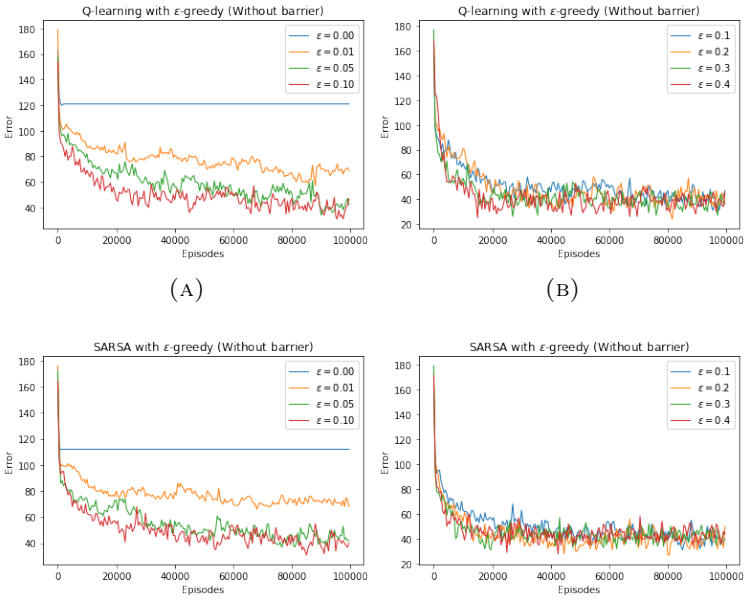
The proof for the convergence of SARSA follows from the convergence of Q learning. The main difference between the convergence of these algorithms is that for SARSA one chooses $F_i(s,a) = r(s,a) + \gamma Q_i(s',b) - Q^\star(s,a)$ with b chosen from a non-greedy policy that in the limit converges to the greedy policy $\pi(s') = \text{argmax}_b\, Q(s',b)$. Both Q learning and SARSA algorithms use sampling, as Monte-Carlo methods do, and bootstrapping, as dynamic programming does. Despite the minuscule difference between both algorithms, in practice, the obtained result can drastically change, as we will see in the following subsection.

1.2.2 Stochastic agent in 2D grid-world pt.2

To exemplify both Q-learning and SARSA algorithms, we use them to solve the same problem proposed in Subsection 1.1.2 of an agent in a 2D grid-world subject to a stochastic wind. We run these algorithms for the same reward maps from Figure 1.1.

Map without barrier

For scenario (A), we obtain the learning curves shown in Figure 1.3 for different behavior ϵ-greedy policies, which consist of choosing the action that maximizes the expected reward with $1-\epsilon$ probability and picking a random action with ϵ probability. We fixed the learning rate with a value of $\alpha = 0.3$ iterated the algorithm for $N = 100,000$ steps. The convergence rate is affected by the exploratory policy (Figure 1.3 (A) and (C)), where taking more random actions allows the agent to explore and update the Q function of more states. Furthermore, increasing exploration does not always mean a decrease in the error as shown in Figures 1.3 (B) and (D). Additionally, we can observe that in the scenario with no barrier, both algorithms converge at similar rates (Figure 1.4). Both algorithms learn a similar policy, and a comparison between the estimated and optimal policies is shown in Figure 1.5. Despite having theoretical convergence, we see that our algorithm does not converge. This happens because the requirements of Theorem 1.2.2 are not satisfied. To further improve the obtained policy, one must decrease the learning rate for better tuning of the Q function as well as increase the iteration steps to get closer to the requirements stated in the convergence theorem.



(A) (B)

(C) (D)

FIGURE 1.3: Error of Q-learning and SARSA algorithms for the MDP problem on Figure 1.1 (A) using various ϵ-greedy behavior policies. The error is calculated by comparing both the estimated policy and the optimal (dynamic programming) policy from Figure 1.2 (A).

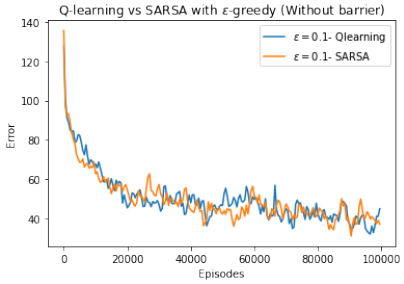
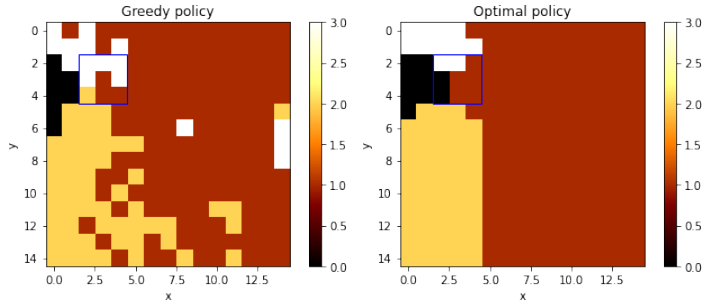


FIGURE 1.4: Comparison of the errors in Q-learning and SARSA algorithms, both with an ϵ-greedy behavior policy, for the problem on Figure 1.1 (A). Each point is a 500 step average of the policy error.



(A) (B)

FIGURE 1.5: Side-to-side comparison between the estimated optimal policy using SARSA with an ϵ-greedy behavior policy with $\epsilon = 0.1$ in Figure (A), and the optimal policy in Figure (B).

Map with barrier

For scenario (B) we proceeded in the same way and with the parameters used in scenario (A). The learning curves are shown in Figure 1.6. For this reward map, an interesting phenomenon occurs; when using SARSA, the error can drastically reduce depending on how much exploration the agent does. The main cause of this phenomenon is the fact that this is an online learning algorithm. Whenever the agent tries to learn an ϵ-greedy policy with a large epsilon, the agent will not take optimal steps since both policies differ by a lot. On the contrary, when the ϵ is low enough to have a near-optimal policy but big enough to explore, the agent learns to surround the obstacle. Comparing the effectiveness of both SARSA and Q-learning (Figure 1.7), we see that at the beginning, SARSA's effectiveness as compared to Q-learning is relatively bad, but after around $30,000$ iterations, the SARSA agent becomes much better than the Q-learning agent. This shows how online and offline learning differs in different settings. After $100,000$ episodes, we compare the policies that were learned by each agent (Figure 1.8). For Q-learning, the agent never learns to surround the obstacle. For SARSA, the agent learns to surround the obstacle. Doing simulations, the drone that follows the obtained policy performs reasonably well and avoids hitting the obstacle most of the time.

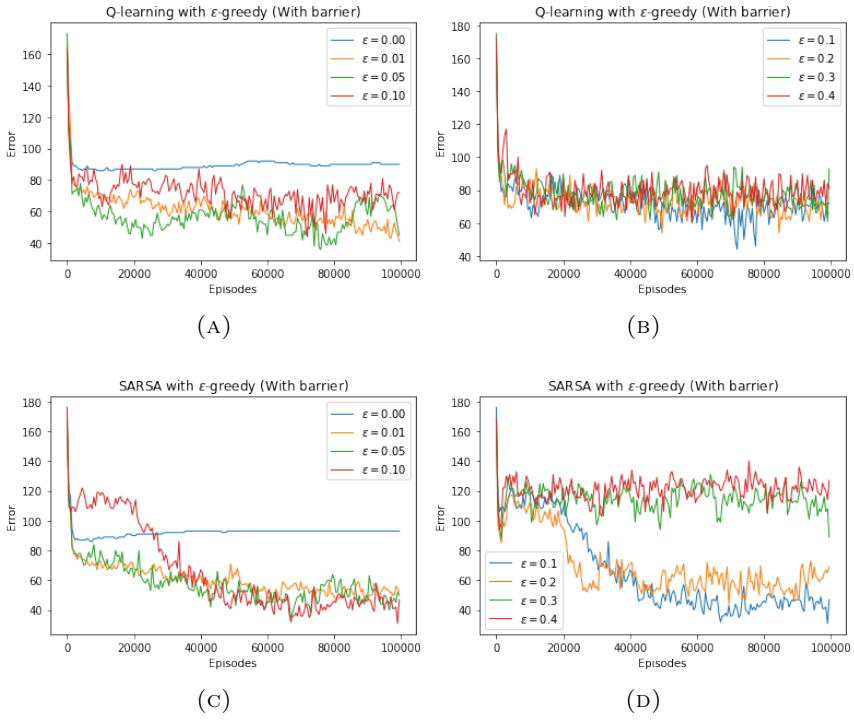


FIGURE 1.6: Error of Q-learning and SARSA algorithms for the 2D MDP problem on Figure 1.1 (B) using various ϵ-greedy behavior policies. The error is calculated in the same way as for the no barrier scenario.

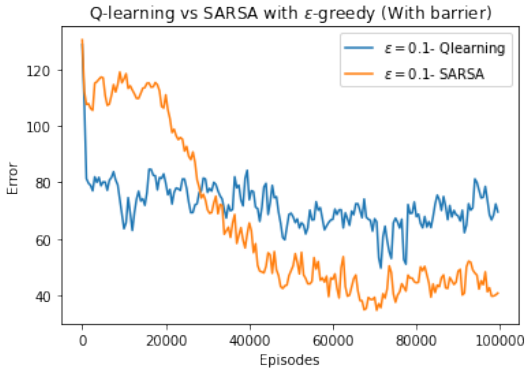
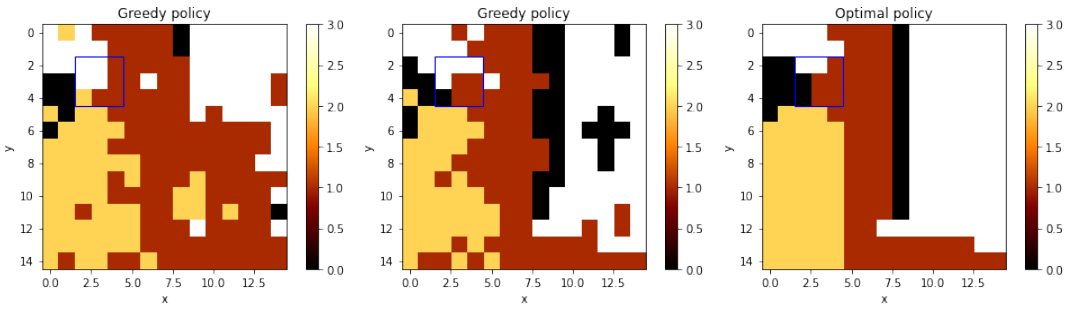


FIGURE 1.7: Comparison of the errors in Q-learning and SARSA algorithms, both with an ϵ-greedy behavior policy, for the problem on Figure 1.1 (B). Each point is a 500 step average of the policy error.



(A) (B) (C)

FIGURE 1.8: Side-to-side comparison between the estimated optimal policy using Q-learning with an ϵ-greedy behavior policy with $\epsilon = 0.1$ in Figure (A), the estimated optimal policy using SARSA with an ϵ-greedy behavior policy with $\epsilon = 0.1$ in Figure (B), and the optimal policy obtained using dynamic programming in Figure (C).

Chapter 2

Neural Networks

This chapter aims to introduce the reader to neural networks (NNs). First, we will introduce the family of functions of NNs and study how they can achieve complex tasks by understanding the intuition behind the universal approximation theorem for a single-layer NN. We will also discuss NN architectures widely used today and how to choose a particular architecture over another. Finally, we will discuss how to train NNs using gradient-based optimization methods and selecting an adequate cost function. We will be following the book by Daniel Roberts and Sho Yaida on Deep Learning Theory [14]. The reader is encouraged to read it if they want a more detailed explanation on this topic.

2.1 Connecting perceptrons

Currently, the world is going through a phase of innovation in artificial intelligence. We see this new phase in the recent technological trends that are rapidly growing, such as self-driving cars, cashier-less checkout grocery stores, recommendation algorithms for social networks, accurate language translator applications, and many more. These trends have achieved what is off today primarily because of neural networks (and hard work). These are tools inspired by the biology of the brain. Neural networks mimic the connection between synapses and their sequential activation by starting from a simple element called a *perceptron*, which we will discuss later in more detail, and making connections between perceptrons to allow mimicking the chain stimulation of some brain synapses.

Mathematically, neural networks are function approximators, just as polynomials are. When we do a Taylor expansion of some well-behaved function f, we do an approximation using multiple powers of x. Suppose we restrict ourselves to some power n of x. In that case, all we have left is a family of polynomials with arbitrary coefficients

$$\mathcal{F} = \left\{ a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n \mid a_i \in \mathbb{R} \right\}.$$

To get an accurate approximation, one must fit the coefficients such that the error of the function approximation under some metric is minimized. In fact, the *Stone–Weierstrass theorem* states that we can get an arbitrary accuracy on approximations of continuous functions using $\mathcal{F}$ in a compact set.

Similarly, neural networks are a family of functions that allow approximating well-behaved functions. The fundamental component of NNs is the perceptron, which is only a fancy name for a linear function (plus a constant) followed by a non-linear function (known as the activation function). Perceptrons of n-dimensional inputs have the form of

$$f\left(w \cdot x + b\right)$$

where $w \in \mathbb{R}^n$, $b \in \mathbb{R}$, and f is an activation function. Some of the commonly used activation functions are:

- the Rectified Linear Unit (ReLu):

$$f(x) = \max(0, x),$$

- the Leaky ReLu:

$$f_a(x) = \max(ax, x),$$

  for $0 < a \leq 1$

- the Sigmoid function:

$$\sigma(x) = \frac{1}{1 + e^x},$$

- the Hyperbolic tangent:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}.$$

This list does not enclose all the used activation functions used nowadays. Now, having defined the perceptron, we can understand what neural networks are. These are a composition of many perceptrons. There are many ways to compose perceptrons, and these give rise to the different architectures discussed up next. NNs approximate other functions by fitting their parameters w and b so that the error under some metric between samples from the original function and the NN's output is minimized.

2.1.1 Multilayer perceptron

The multilayer perceptron (MLP) is the simplest non-trivial neural network one can think of. It consists of multiple layers of fully connected perceptrons. The MLP is crafted using a non-linear function f for each perceptron and making linear combinations using the weights W_{ij} and bias terms b_i for each perceptron i on layer m. The output of each layer is defined recursively:

$$z_i^{(1)}((x_{1,k}, \ldots, x_{n_0,k})) = b_i^{(1)} + \sum_{j=1}^{n_0} W_{ij}^{(1)} x_{j,k}, \quad \forall i \in \{1, \ldots, n_1\},$$

$$z_i^{(m)}((x_{1,k}, \ldots, x_{n_0,k})) = b_i^{(m)} + \sum_{j=1}^{n_m} W_{ij}^{(m)} f(z_j^{(m-1)}((x_{1,k}, \ldots, x_{n_0,k}))), \quad \forall i \in \{1, \ldots, n_m\},$$

where $(x_{1,k}, \ldots, x_{n_0,k})$ is a the k-th input vector of size n_0. An example of an MLP with two hidden layers is shown in Figure 2.1. The number of nodes on a specific layer is the dimension of some feature, in this example, we embed a 5 dimensional vector in $\mathbb{R}^7$, then into $\mathbb{R}^4$, and the output is a consequently embedding in $\mathbb{R}^3$.

MLPs have great expressivity, meaning that they can accurately approximate many continuous functions. Consequently, one of the first use cases of MLPs was to generate low-dimensional encoders and decoders, known as auto-encoders, similar to what principal component analysis achieves but in a non-linear manner. For example, for a data set of k features with possible correlations, one could construct a NN architecture with k input nodes, then one hidden layer with $m < k$ nodes, and finally another layer with k nodes. Successfully training such a neural network with a loss function that penalizes differences between the inputs and the outputs would give rise

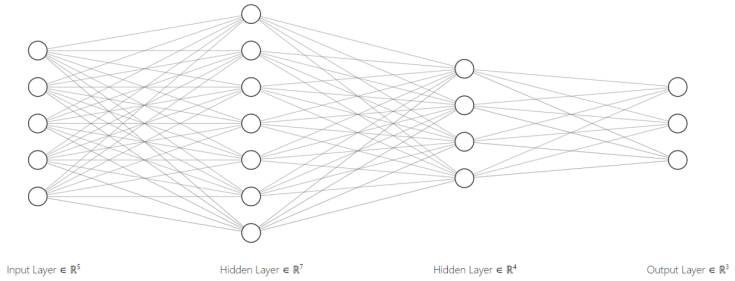


FIGURE 2.1: Example of a MLP with 4 layers that has a five-dimensional input and a three-dimensional output.

to an encoder and decoder (we will discuss training later in the chapter). First, the k features are encoded into m features using the two first layers. Second, we can obtain a decoder from the low dimensional space of m features to the original k features using the last two layers of the NN. For getting a more complex and powerful auto-encoder, the idea is the same. One passes data through a set of hidden layers until reaching the desired dimension and then mirrors the constructed neural network, such as the network shown in Figure 2.2.

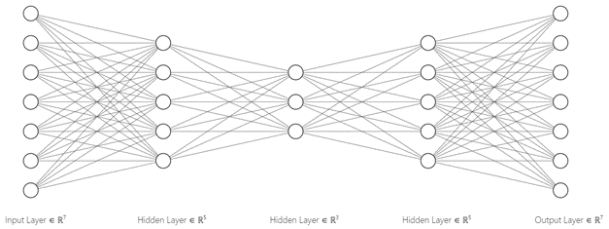


FIGURE 2.2: An example of an autoencoder with input dimension of 7 and encoding dimension of 3.

Let us now prove how these families of functions encapsulated in neural networks are so good at approximating other functions with the following simplified version of the universal approximation theorem for neural networks by Cybenko [6]:

Theorem 2.1.1. *The set of finite linear combinations of perceptrons with a sigmoid activation function*

$$\left\{ \sum_{i=0}^{N} a_i \sigma(\omega \cdot x + b) \;\middle|\; a_i \in \mathbb{R} \right\}$$

is dense in $C([a,b])$.

Proof. Let $f \in C([a,b])$ and $\epsilon > 0$. Let us find a function $\sum_{i=0}^{N} a_i \sigma(\omega \cdot x + b)$ that arbitrarily approximates f. Let $g_\lambda(x) = \sigma(\lambda(\omega x + \theta))$, when $\lambda \to \infty$ we get $g_\lambda(x) \to H_\ell(x)$, where $\ell = \frac{\theta}{\omega}$ and $H_\ell(x)$ is the Heaviside function defined by

$$H_\ell(x) = \begin{cases} 0, & x \leq \ell \\ 1, & x > \ell \end{cases}.$$

Then, we can construct by brute force a function that approximates f that is a linear combination of perceptrons with a Heaviside activation function as follows:

$$h(x) = \sum_{n=0}^{\lfloor \frac{b-a}{\epsilon} \rfloor} f(n\epsilon + a) \left[H_{n\epsilon+a}(x) - H_{(n+1)\epsilon+a}(x) \right]$$

Therefore, the error between both h and f

$$\begin{aligned} \mathcal{E} &= \int_a^b |f(x) - h(x)| dx \\ &\leq \left(\int_a^b |f(x)| dx - \sum_{n=0}^{\lfloor \frac{b-a}{\epsilon} \rfloor} |f(n\epsilon + a)| \epsilon \right) \xrightarrow[\epsilon \to 0]{} 0 \end{aligned}$$

since f is Riemann-integrable because it is continuous in a closed interval and we are subtracting the Left Riemman sum from it. □

This result can be extended to functions in $C([a, b]^n)$ and neural networks with different activation functions such as ReLu's and others. The proof of Theorem 2.1.1 does not seem very practical since it is based on a single layer network and it requires many perceptrons to achieve an accurate approximation, but in practice, when we include multiple layers, the expressivity of these networks increase significantly compared to single-layer networks. We will discuss an example of this behavior in the following subsection.

2.1.2 Convolutional Neural Networks (and more)

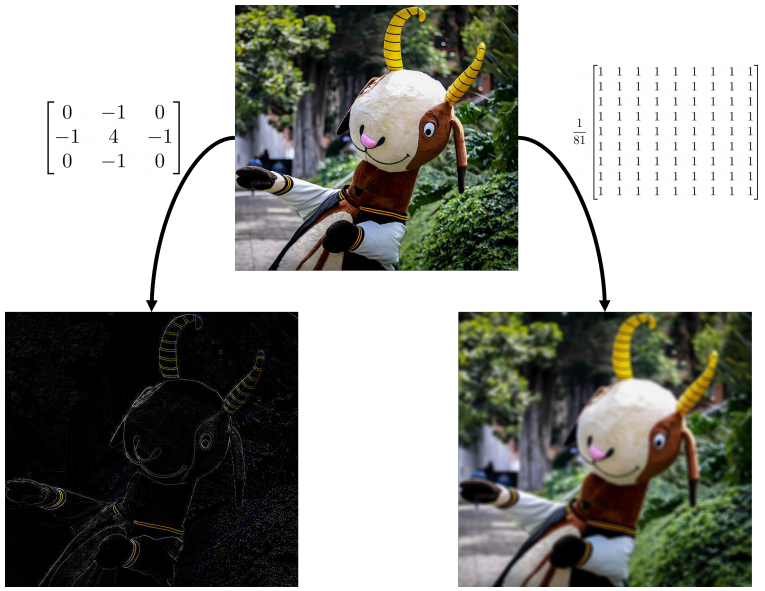


FIGURE 2.3: A picture with size $(600, 640)$ after the convolution of stride 1 with two filters that generate edge detection and blur, respectively.

Convolutional neural networks (CNNs) are a subset of MLPs, as they are layered perceptrons with local connections. CNNs have shown excellent capabilities for image processing tasks, such as object classification and facial recognition. This architecture exploits the data structure to reduce the problem's dimensionality and helps find well-performing neural networks compared to MLPs. The inspiration of CNNs is image

processing kernels as the ones shown in Figure 2.3. These kernels are matrices W_{ij} that are used to modify an image in some desired way. They work by changing a pixel's value by the value obtained from point-wise multiplying the surrounding pixels with the kernel, also known as doing a convolution between the kernel and the image. This process can be thought of as a small window that generates another image by sweeping the original image and multiplying all the pixels inside the window by some weights $W_{i,j}$. Kernels can vary in size, and we can sweep them in different pixel intervals. The number of pixels that the filter moves with each sweep is called the *stride*, so a filter with stride 1 sweeps all the image (excluding the borders), and with stride 5 it sweeps the image by skipping 4 pixels each step.

CNNs are neural networks on which the parameters to be updated are the weights (or entries) of image processing kernels towards minimizing some cost function. This idea restricts the number of connections of the neural network and, subsequently, the number of parameters required between layers. A CNN consists of multiple layers containing different filters, each with its own weights $W_{i,j}$. Therefore, for a given activation function f, the CNN layer outputs are defined recursively as:

$$z^{(1)}_{i,(a,b)}(x_\alpha) = b^{(1)}_i + \sum_{j=1}^{n_0} \sum_{a'=-k}^{k} \sum_{b'=-k}^{k} W^{(1)}_{ij} f\left(z^{(0)}_{j,(a+a',b+b')}(x_\alpha)\right), \quad \forall i \in \{1, \dots, n_1\},$$

$$z^{(m)}_{i,(a,b)}(x_\alpha) = b^{(m)}_i + \sum_{j=1}^{n_{m-1}} \sum_{a'=-k}^{k} \sum_{b'=-k}^{k} W^{(m)}_{ij} f\left(z^{(m-1)}_{j,(a+a',b+b')}(x_\alpha)\right),$$

where $z^{(m)}_{i,(a,b)}$ refers to the pixel (a, b) from filter i in layer m and x_α is the input of the CNN (a matrix of the pixel values of an image). An example of a CNN that could be used for feature extraction is shown in 2.4.

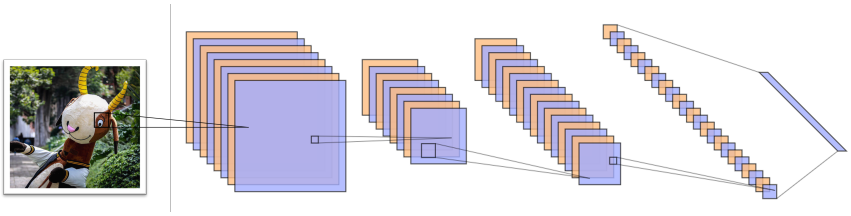


FIGURE 2.4: Example of a convolutional neural network with a fully connected layer as the output layer.

Some specific machine learning problems require convolutional neural networks to detect image features that are complicated to specify but have a simple task. For these problems, CNNs can be mixed with fully connected neural networks, and we train the NN as a whole instead of training the CNN to capture features on a NN separately and afterward processing them with an MLP. An example where we would like to do so is determining if a collection of electron spins are in either a ferromagnetic or paramagnetic phase of matter [15]. There are no apparent image features to extract here, but the answer is still a simple classification between two possibilities. An example where you would not like training the NN from beginning to end would be autonomous driving. In such a sensitive case, one would like to identify pedestrians from cars and fully control each NN layer. In either case, CNNs and NNs are no solution for every task, which is why there are so many neural network architectures. For completeness, let us mention some other relevant NN architectures.

Other architectures

All (feed-forward) neural networks are subsets of fully connected MLPs, so a natural question to ask is why it is worth it considering other particular architectures of neural networks. It is unfeasible to train very large MLPs and make them excel at specific and delicate tasks, such as speech recognition, language translation, object detection in images, and more. We already discussed CNNs handle tasks related to pictures. Still, we need different architectures for sequential data, like the market value of some stock or the sentence translation between two languages. For these tasks, three main NN architectures are widely used:

- Recurrent Neural Networks (RNN): As the name says, RNNs are neural networks used for recurrent tasks. An example is word prediction. Imagine you have an incomplete sentence, and you want to finish it with one last word:

$$\underbrace{\text{"Seneca is a recognized Stoic}}_{\text{input}}\ \underbrace{\text{philosopher"}}_{\text{predict}}.$$

  If we used an MLP, we would have a fixed input size, and using the incomplete sentence as the input of the MLP would be a problem due to the varying lengths of different sentences. This inspired the RNN architecture, where a network has a hidden variable that is passed through time on every iteration:

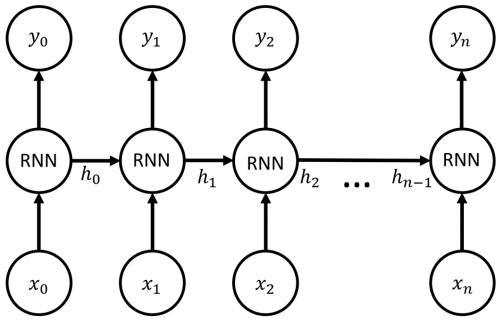


FIGURE 2.5: Standard architecture of a recurrent neural network.

  Where the hidden state is updated as $h_{i+1} = \tanh(W_1 x + W_2 h_i)$ and the output vector is $y_i = W_3 h_i$. The most relevant problem that this kind of NN presents is the exploding and vanishing of the gradient after several iterations. Thus, a different recurrent cell that can track long-term and short-term correlations should be used. These cells are known as gated cells, and there are several of them, with the Long-Short Term Memory (LSTM) [16] being one of the most popular cells. The main difference between these more complex recurrent cells and the original recurrent cell is that the insides of the former are designed to have an uninterrupted gradient flow achieving better results than the latter.

- Transformer Networks [17]: The preferred mechanism was recurrent neural networks when dealing with sequential transductive inference; this means reasoning from some training cases to test cases for sequential data. For a task such as language translations, the RNN architecture was used to encode a sentence and later decode such sentence in the preferred language. A problem RNNs faced was that the last hidden state h_i from the encoder did not give enough context to the decoder to do a correct translation. It would be like if you were asked

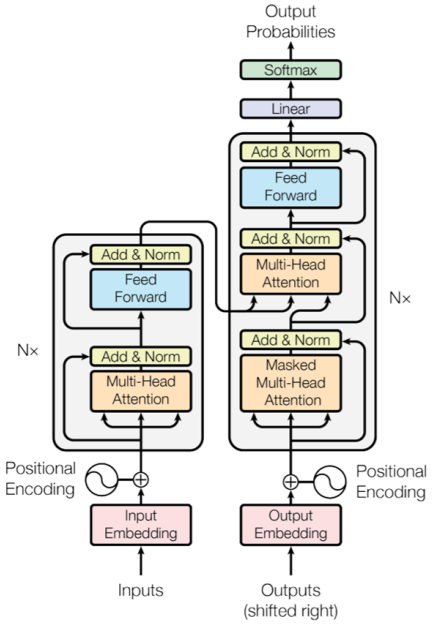


FIGURE 2.6: The original transformer model. Taken from [17].

to translate this Chapter to Spanish right after reading it. The way to do so would be by writing down keywords that give you some context to perform this task. This concept is the main idea behind attention mechanisms. RNNs were enhanced by using attention in which the hidden states of each step were saved to feed into decoding RNNs. Recently, it was shown that a new model known as Transformers could solve these tasks without having to appeal to recurrent architectures [17]. The originally proposed architecture of the transformers does not use any recurrent neural network as shown in Figure 2.6. This model is extremely powerful and has shown amazing performance in natural language processing models such as GPT-3 [18] and OpenAI Codex [19], and in computer vision with Vision Transformers (ViT) [20]. These models have been recently used for Reinforcement Learning [21], but they have not yet shown superior performance than standard RL algorithms. Still, this model opens a new and promising research area yet to be explored.

2.1.3 Training NNs

Neural networks can have many parameters, ranging from hundreds to hundreds of billions. Finding the best choice of parameters of a NN can be challenging, and the way it is generally done is using the *backpropagation algorithm*. To understand this algorithm, we must first understand what fitting the weights of a NN means. Similar to linear regression, where one finds a plane that best fits a data set under some norm, typically the mean square error, for NNs, the idea is the same. We need to pick some norm to measure the error of the network with respect to a data set and then optimize the network's parameters to minimize such error. As for optimizers, due to the strong non-linearity of these functions, the general approach is to do gradient-based optimization. Let us break down these main points:

Cost functions

Under the paradigm of supervised learning, the performance of a particular neural network architecture begins with the correct choice of a cost function. There are a great variety of cost functions to measure the predictive performance of a NN for a labeled data set with N elements, but among the most popular ones, the followings stand out:

- Mean squared error (MSE):

$$\mathcal{L}_{MSE} = \frac{1}{2}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 .$$

  where y_i is the label and $\hat{y}_i$ is the predicted value. This is one of the most common cost functions used to train neural networks. Note that for gradient-based algorithms, the mean absolute error (MAE) defined by $\sum_{i=1}^{N}|y_i - \hat{y}_i|$ would be a bad choice due to the non-differentiability of the absolute value function.

- Huber loss:

$$\text{Let } \delta > 0, \quad C_\delta(y, \hat{y}) = \begin{cases} \frac{1}{2}(y-\hat{y})^2 & \text{for } |y-\hat{y}| \leq \delta \\ \delta\left(|y-\hat{y}| - \frac{1}{2}\delta\right) & \text{otherwise} \end{cases}$$

$$\mathcal{L}_{HL(\delta)} = \sum_{i=1}^{N} C_\delta(y_i, \hat{y}_i)$$

  We can mix both the MSE and the MAE to get an outliers robust cost function. We define the Huber loss as using the MSE near the non-differentiability point of the MAE. This loss function allows getting mean and median unbiased estimators.

- Binary cross-entropy:

$$\mathcal{L}_{CE} = -\sum(y_i \log(p_i) + (1-y_i)\log(1-p_i))$$

  This cost function is used in settings where the output p of a neural network is between 0 and 1, and the problem consists of binary classification of data points. The main difference between MSE and cross-entropy is that the missclassification of some input is highly penalized in the cross-entropy cost function compared to MSE.

There are many other loss functions and variations of the ones mentioned, such as adding regularization, which prevents overfitting. In practice, for high accuracy in specific tasks, the cost function can be much more complex to achieve the desired behavior of the neural network. The cost functions' outputs will vary for different weights of some NN, and the goal of training is finding the global minimum of this non-linear landscape. To do so, we need to take gradients of neural networks and proceed by gradient-descent-like optimizers.

Gradients with backpropagation

Perceptrons are linear functions followed by a non-linear function with a defined derivative. Thus, we simply need to differentiate the non-linear function and multiply it by some constant to find the derivative for some parameter. As for neural

networks, they are just concatenations of perceptrons. We can use the Leibniz rule, also known as the product rule, to get the derivative of the neural network with respect to some particular weight.

Let us illustrate these calculations for an MLP. The derivative of the k-th component of the output vector $z^{(L)}$ of some MLP neural network with respect to some weight $W_{ij}^{(m)}$ from layer m is

$$\frac{dz_k^{(L)}}{dW_{ij}^{(m)}} = \sum_{p=1}^{n_m} \frac{dz_k^{(L)}}{dz_p^{(m+1)}} W_{pi}^{(m+1)} f'(z_i^{(m)}) f(z_j^{(m-1)})$$

and with respect to some bias term $b_i^{(m)}$ from layer m is

$$\frac{dz_k^{(L)}}{db_i^{(m)}} = \sum_{p=1}^{n_m} \frac{dz_k^{(L)}}{dz_p^{(m+1)}} W_{pi}^{(m+1)} f'(z_i^{(m)}).$$

From these equation we can see that, unless $m = L-1$, we must recursively calculate

$$\frac{dz_k^{(L)}}{dz_p^{(m)}}$$

to obtain an expression in terms of derivatives from the next layer

$$\frac{dz_k^{(L)}}{dz_p^{(m+1)}}.$$

This is obtained by applying chain rule over each layer. Thus, propagating and multiplying all gradients backwards from the last layer L to the layer of interest m. An interesting geometrical consequence of chain rule is that all the terms that are obtained expanding

$$\frac{dz_k^{(L)}}{dz_p^{(m)}}$$

are the sums of the product of the derivatives along all possible one-way paths connecting the feature $z_k^{(L)}$ with the feature $z_p^{(m)}$. Having this calculation, we now understand where the name *backpropagation* comes from and are ready to use this technique to optimize the weights of a neural network.

Optimizers

Optimization of non-linear functions is a well-studied topic, and many algorithms accomplish this task, one of them being gradient descent (GD). This method and other gradient-based algorithms are the default methods of optimizing the parameters of a neural network. This is due to the effectiveness of gradient-based algorithms in practice. Since training a network aims to find the best parameters that achieve the lowest value of the cost function $f(x)$, a non-linear function, we can do this by iteratively using gradient descent to update such parameters:

- Gradient descent (GD):

  Given a particular loss function $\mathcal{L}$, the vector $-\nabla_\theta \mathcal{L}$ points in the direction of maximum descent of $\mathcal{L}$, thus, updating the parameters as

$$\vec{\theta} \leftarrow \vec{\theta} - \eta \nabla_\theta \mathcal{L}(\theta)$$

for some sufficiently small $\eta \in (0, 1]$ will decrease the loss function value. For MSE, this vector is given by

$$\nabla \mathcal{L}_{MSE} = \sum_{i=1}^{N} \frac{1}{2} \nabla_\theta (y_i - \hat{y}_i)^2 = \sum_{i=1}^{N} (y_i - \hat{y}_i) \nabla_\theta (y_i - \hat{y}_i).$$

One of the problems of obtaining this vector is that the gradient needs to be taken for every sample $i \in \{1, \ldots, N\}$ and this becomes computationally expensive for $N >> 1$. Additionally, unless the loss function is convex, convergence is not guaranteed due to possible local minima.

- Stochastic batch gradient descent (SGD) [22]:

  Similar to gradient descent, SGD uses the same update rule to modify the model parameters, with the small difference of using an incomplete loss function that only takes into account partial data points instead of the complete data-set. This is done to increase efficiency. This algorithm, despite having higher variance than GD, has been shown to converge as GD would do [22]. Having the update rule

  $$\vec{\theta} \leftarrow \vec{\theta} - \eta \nabla \mathcal{L}_{batch}(\theta),$$

  and loss function

  $$\mathcal{L}_{MSE(batch)} = \frac{1}{2} \sum_{i=1}^{M} (y_i - \hat{y}_i)^2, \quad 0 < M < N,$$

  the vector $\nabla \mathcal{L}_{MSE(batch)}$ can be calculated efficiently independent of the data-set size.

- Gradient descent with momentum:

  Generally, GD and SGD suffer from getting stuck in a local minima of the loss function landscape. A way of avoiding this is by adding a term called momentum, inspired from the momentum notion in physics, in which the update rule is not only affected by the previous step but geometrically on all past vectors:

  $$v \leftarrow \gamma v + \eta \nabla_\theta \mathcal{L}(\theta)$$
  $$\theta \leftarrow \theta - v,$$

  with $\gamma \approx 0.9$. There are some momentum-based optimizers that intend solving different convergence problems as well as speed up the convergence of the optimization. A popular momentum-based method is the Nesterov gradient descent [23], in which the update rule is the same as GD with momentum, but the gradient of the loss function is evaluated shifting the parameters as $\nabla_\theta \mathcal{L}(\theta - \gamma v)$ rather than $\nabla_\theta \mathcal{L}(\theta)$.

- Adaptive moment estimation (Adam) [24]:

  One of the most popular optimization method for neural networks is the Adam optimizer. This method uses momentum techniques, keeps track of past gradients, and corrects for biases in momentum calculations for obtaining one of the

best performing optimization methods for current models. The update rule for the neural network's parameters is the following:

$$
\begin{aligned}
i &\leftarrow i + 1 \\
m &\leftarrow \alpha m + (1 - \alpha)\nabla_\theta \mathcal{L} \\
v &\leftarrow \beta v + (1 - \beta)\left(\nabla_\theta \mathcal{L}\right)^2 \\
\theta &\leftarrow \theta - \frac{\eta}{\sqrt{\frac{v}{1-\beta^i}} + \epsilon}\left(\frac{m}{1-\alpha^i}\right)
\end{aligned}
$$

where $\alpha \approx 0.9$, $\beta \approx 0.999$, and $\epsilon \approx 10^{-4}$. These hyper-parameters can affect the convergence of the algorithm. In real life applications, these and many other hyper-parameters are empirically explored to obtain better results.

Chapter 3

Deep Q-learning

This chapter will introduce the deep q-learning algorithm (DQN) and discuss different aspects of its convergence. Additionally, we will examine some variants of this algorithm, such as Double DQN and Dueling DQN. Finally, we will implement the DQN algorithm to solve three different Atari games using the OpenAI gym and Keras-RL libraries. The code used in this chapter can be found on this link.

3.1 DQN agents

The DQN algorithm is an approach to solving a high-dimensional reinforcement learning problem using neural networks to approximate the Q value function of the corresponding MDP. At first sight, it might seem wrong since neural networks, as we discussed in the previous chapter, are used in the supervised learning paradigm, and there are no datasets or labels in an MDP problem. However, we can shift this paradigm to reinforcement learning by storing states and rewards in a data set (known as *replay buffer*) and training a neural network to predict an expected reward. This shift is the main idea behind the DQN algorithm. Still, there are many caveats on doing so, such as managing correlated input states and obtaining stability of the NNs, and these are the main points discussed in this section.

3.1.1 (Vainilla) DQN algorithm

There have been multiple proposals for merging both the fields of reinforcement learning and deep learning that were not practical for solving complex tasks due to various reasons, such as instability of the neural network's weights or correlations between data used for training. Recently, the Deepmind group [7] proposed a way of using neural networks for complex reinforcement learning tasks, in particular, playing Atari games. This algorithm is known as the *DQN agent*, and its implementation is shown in Algorithm 5. It is worth noting that the way states are stored in Algorithm 5, where the oldest data point is discarded after the replay buffer is full, helps to avoid storing useless data after several training episodes have passed. This is because the neural network will begin to fine-tune in latter episodes with more precise and accurate, rather than random, state-action pairs. Additionally, to avoid instability in training, two procedures are done. First, the data is sampled from the memory in a random way so that the training data is uncorrelated. And second, the target $(\gamma \max_b Q(s', b, \theta'))$ is estimated using old weights θ' compared to the weights θ used to choose the action. These two procedures are key to getting a trainable DQN agent. In practice, without implementing them, the optimization of the weights will be much harder (this is something we will discuss in further detail).

This algorithm has been used in several disciplines with complex tasks, such as algorithmic trading, autonomous driving, classical and quantum optimal control, and

Algorithm 5 DQN with an ϵ-greedy policy [7]

Let M be a memory (replay buffer) able to store N states
Let $Q(\theta)$ be a state-action NN with random weights θ
Let $\theta' \leftarrow \theta$
for $i \in \{1, \cdots, M\}$ **do**
  for $j \in \{1, \cdots, \tau\}$ **do**
    Let s be the current state.
    if $\text{rand}(0, 1) < \epsilon$ **then**
      $a = \text{random\_choice}(\mathcal{A}_s)$
    else
      $a = \text{argmax}_a \, Q(s, a, \theta)$
    end if
    Let s' and r be the next state and obtained reward, respectively.
    Store (s, a, r, s') in M
    if s' is not a terminal state **then**
      $y \leftarrow r + \gamma \max_b Q(s', b, \theta')$
    else
      $y \leftarrow r$
    end if
    $\mathcal{L}(\theta) \leftarrow \hat{\mathbb{E}}\left[(r + \gamma \max_{a'} Q(s', a', \theta') - Q(s, a, \theta))^2\right]$ calculated sampling M.
    $\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}(\theta)$
    if $j = 0 \pmod k$ **then**
      $\theta' \leftarrow \theta$
    end if
  end for
end for

many more problems. It would seem that it works like magic whenever you have a problem, and the agent will learn to solve it. However, this is not the case for most tasks (that can be formalized as MDPs) given a random choice of hyper-parameters, such as NN architecture, memory size, behavior policy, etc. Hyper-parameters must be chosen in a particular way that highly depends on the MDP that we are trying to solve, and when done correctly, a great capability could be achieved.

Despite the high performance of this algorithm in several tasks, it is not guaranteed that it converges to the optimal policy due to the non-linear landscape of the cost function of the neural network. We will try to understand the convergence of the DQN agent to the optimal policy $Q^\star$ in the following subsection.

3.1.2 Convergence bounds

Recently, Fan et al. [10] proposed to understand the convergence of the DQN algorithm by studying a simplified version of DQN known as the fitted Q-iteration algorithm [25]. The only difference between this algorithm and DQN from Algorithm 5 is the lack of a replay buffer M to store state-action-reward-state tuples. In the former, the tuples are obtained from a sampling distribution σ. In contrast, the latter approximates this sampling distribution using the memory, meaning that a larger memory will lead to a more accurate representation of σ.

The main result about the convergence of the fitted Q-iteration algorithm requires a few assumptions, but before mentioning them, let us define different families of neural networks which are important for this result:

Definition 3.1.1. The family of sparse ReLu networks is defined as

$$\mathcal{W}_{L,s,d_j} = \left\{ f \text{ NN with ReLu} : \max_{l\in\{0,\dots,L+1\}} \left\|\vec{\theta}_l\right\|_\infty \leq 1, \sum_{l=1}^{L+1} \left\|\vec{\theta}_l\right\|_0 \leq s, \max_{i\in\{d_0,\dots,d_{L+1}\}} \|f_i\|_\infty \leq V \right\}$$

where L is the number of hidden layers, s is the sparsity, d_j is the width of the j-th layer, $V \in (0,\infty)$, and $\vec{\theta}_l$ are all the weights and bias of the layer l. Here, we restrict the maximum weight to be 1 and the number of important neurons to be s.

Definition 3.1.2. The set of Hölder functions over a compact $C \subseteq \mathbb{R}^n$ is defined by

$$\mathcal{D}_{\beta,K} = \left\{ f : C \to \mathbb{R} : \sum_{\alpha:|\alpha|<\beta} \|\partial^\alpha f\|_\infty + \sum_{\alpha:|\alpha|_1=\lfloor\beta\rfloor} \sup_{\substack{x,y\in\mathcal{C}\\ x\neq y}} \frac{|\partial^\alpha f(x) - \partial^\alpha(y)|}{\|x-y\|_\infty^{\beta-\lfloor\beta\rfloor}} \leq K \right\},$$

where $\beta, K > 0$ and $\partial^\alpha = \prod_{i=1}^n \partial^{\alpha_i}$. We will denote by $\mathcal{G}_q$ all the valid[1] compositions of q Hölder functions.

To understand the use of this set, note that Hölder functions are a generalization of Lipschitz functions, allowing a wider set of continuous functions for this analysis. Furthermore, to define the family of all possible neural network Q-functions with ReLu activation function, consider the following classes of functions:

Definition 3.1.3. The extended sparse ReLu networks $\widetilde{\mathcal{W}}_{L,s,d_j}$ and the extended composition of Hölder functions $\widetilde{\mathcal{G}}_q$ are defined as

$$\widetilde{\mathcal{W}}_{L,s,d_j} = \left\{ f : \mathcal{S}\times\mathcal{A} : \forall a \in \mathcal{A}, f(\cdot,a) \in \mathcal{W}_{L,s,d_j} \right\},$$
$$\widetilde{\mathcal{G}}_q = \left\{ f : \mathcal{S}\times\mathcal{A} : \forall a \in \mathcal{A}, f(\cdot,a) \in \mathcal{G}_q \right\}.$$

Having defined these sets, we will be assuming that our Q-networks are elements of $\widetilde{\mathcal{G}}_q$. The assumption of using sparse ReLu neural networks relies on the Lottery Ticket hypothesis [26]. In a few words, it says that one can find a sparse neural network f_s, a subnetwork of a dense network f such that the performance (cost function) of f_s is close to that of f.

Furthermore, for Theorem 3.1.1 to be valid, the following two assumptions were made:

- For all $f \in \widetilde{\mathcal{W}}_{L,s,d_j}$, we have that

$$r(s,a) + \gamma\mathbb{E}\left[\max_{a'\in\mathcal{A}} Q(S',a) \mid S' \sim P(s,a)\right] \in \widetilde{\mathcal{G}}_q.$$

- Let $\nu_1, \nu_2 \in \mathcal{D}[\mathcal{S}\times\mathcal{A}]$ two absolutely continuous probability measures and $\mu = (\mu_1, \cdots, \mu_i, \cdots)$ a policy for the MDP. Denote $P_T^\mu \nu_1 = P^{\mu_T} \cdots P^{\mu_1}\nu_1$ the probability distribution of states $\{(S_i, A_i)\}_{i=1,\dots,T}$ of the MDP that has $(S_0, A_0) \sim v_1$. Additionally, the j-th κ concentration coefficient is defined by

$$\kappa(j,\nu_1,\nu_2) := \sup_\mu \mathbb{E}_{\nu_2}\left[\left|\frac{dP^\mu\nu_1}{d\nu_2}\right|^2\right].$$

[1] Meaning that the range of f_i is the domain of f_{i+1}.

where $\frac{dP^{\mu}\nu_1}{d\nu_2}$ is the Radon–Nikodym derivative. Then, we assume that there exist a coefficient $\phi(\nu, \sigma)$ such that

$$(1-\gamma)^2 \cdot \sum_{j=1}^{\infty} \gamma^{j-1} \cdot j \cdot \kappa(j, \nu, \sigma) \leq \phi(\nu, \sigma)$$

where σ is the sampling distribution of the fitted Q-iteration algorithm and ν is an initialization distribution.

Under the above conditions, the following theorem holds:

Theorem 3.1.1 (Fan, et al. (2020)[10])**.** *Let π_t be the greedy policy in step t of the fitted Q-iteration algorithm. There exists a constant $C \in \mathbb{R}^+$ such that*

$$\|Q^{\star} - Q^{\pi_t}\|_{1,\nu} \leq \frac{C}{(1-\gamma)^2} \phi(\nu, \sigma) \gamma |\mathcal{A}| (\ln n)^{1+2\xi} n^{(\alpha-1)/2} + \frac{4\gamma^{t+1} R_{\max}}{(1-\gamma)^2}$$

where $\alpha = \max_{j=1,\ldots,q} \frac{t_j}{2\beta_j \prod_{l=1}^{j+1} \min(1, \beta_l) + t_j}$, β_j is the constant defining the j-th Hölder functions in the composition of q Hölder functions, n is the sample size, t_j is the maximum number of input variables on which a j-th Hölder function depends, and ξ satisfies

$$\max \left\{ \sum_{j=1}^{q} (t_j + \beta_j + 1)^{3+t_j}, \sum_{j=1}^{q} \ln(t_j + \beta_j), \max_{j \in \{1,\ldots,q\}} p_j \right\} \leq \tilde{C} (\log n)^{\frac{\xi - 1}{2}}$$

for some $\tilde{C} > 0$, and p_j the dimension of the domain of f_j, the j-th Hölder function.

The two terms in this bound correspond to: first, a statistical error:

$$\frac{C}{(1-\gamma)^2} \phi(\nu, \sigma) \gamma |\mathcal{A}| (\ln n)^{1+2\xi} n^{(\alpha-1)/2}$$

and second, a algorithmic error

$$\frac{4\gamma^{t+1} R_{\max}}{(1-\gamma)^2}$$

that decreases by a factor of γ every iteration t. Thus, for large t, where the statistical error is dominant, the error rate achieved by the fitted Q-iteration algorithm will be

$$\|Q^{\star} - Q^{\pi_t}\|_{1,\nu} \propto |\mathcal{A}| (\ln n)^{1+2\xi} n^{(\alpha-1)/2}.$$

3.2 DQN variants

In practice, the DQN agent from algorithm 5 could be slightly tweaked to obtain (in some cases) better performance. The two main variants are the double DQN and the dueling DQN algorithms, which we will discuss in this section.

3.2.1 Double DQN

The double DQN algorithm is a DQN variant inspired by the double Q-learning variant of table Q-learning, where two q-tables (A and B) are used to update the state-action function rather than just one. In this table version, double q-learning updates A (B) values using a target calculated using B (A). This means the update step is given by

$$Q_A(s,a) \leftarrow (1-\alpha)Q_A(s,a) + \alpha(r + \gamma Q_B(s', \text{argmax}_a\, Q_A(s',a)),$$

and similarly for table B. Likewise, double DQN uses two different neural networks to pick an action and to estimate the target value. Having this small change implemented, the new procedure is shown in Algorithm 6.

Algorithm 6 Double DQN [27]

```
Let M be a memory able to store N states
Let Q(θ) be a state-action NN with random weights θ
Let T(θ') be the target state-action NN with weights θ' = θ
for i ∈ {1, ··· , M} do
  for j ∈ {1, ··· , τ} do
    Let s be the current state.
    if rand(0, 1) < ϵ then
      a = random_choice(A_s)
    else
      a = argmax_a Q(s, a, θ)
    end if
    Let s' and r be the next state and obtained reward, respectively.
    Store (s, a, r, s') in M
    if s' is not a terminal state then
      y ← r + γ max_b T(s', b, θ')
    else
      y ← r
    end if
    θ ← θ − (η/2) ∇_θ (y − Q(s, a, θ))^2
    if j = 0 (mod k) then
      θ' ← θ
    end if
  end for
end for
```

A benefit from this variant is a better estimate of the true $Q(s,a)$ values. It was shown in [27] that, given some unbiased estimates $Q_t(s,a)$ such that

$$\sum_a (Q_t(s,a) - V^\star(s)) = 0, \quad \frac{1}{N}\sum_a (Q_t(s,a) - V^\star(s))^2 = k > 0$$

the $Q_t(s,a)$ estimates have a strict lower bound given by

$$\max_a Q_t(s,a) - V^\star(s) \geq \sqrt{\frac{k}{N-1}}$$

whilst the double Q-learning estimates have a zero-valued lower bound

$$\left|Q_t^A(s, \text{argmax}_a\, Q_t^B(s,a)) - V^\star(s)\right| \geq 0.$$

This lower bound makes a DQN agent sometimes overestimate the actual value of a Q value function. We will see in the implementation section that double DQN has lower predicted values for Q as a consequence of this result.

3.2.2 Dueling DQN

This is another architecture proposed for DQN. It was proposed as a way of improving the DQN agent, making it more appropriate for model-free reinforcement learning. For the dueling network [28], the way of predicting the state-value function is different. The idea now is to calculate separately the value functions, something familiar to us, an the advantage function defined by:

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s).$$

The network used for this task is shown in Figure 3.1. Having calculated both V and A, we can recover the state-value function in different ways. For example:

$$Q(s,a) = V(s) + A(s,a) - \max_b A(s,b),$$

or

$$Q(s,a) = V(s) + A(s,a) - \frac{1}{|\mathcal{A}|}\sum_b A(s,b).$$

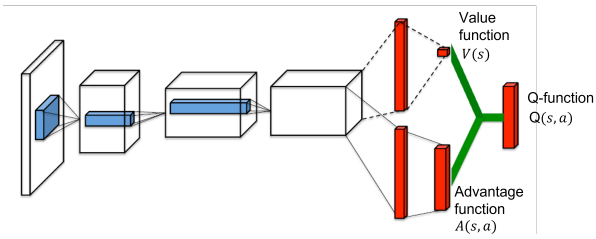


FIGURE 3.1: Dueling DQN network architecture. Modified from [28].

The main innovation of this new architecture is that, given a state s, we can calculate the value of that state without explicitly caring about the effect of any action over the state. This could be useful whenever the action taken does not change the environment in a significant manner. The training method for such a network is the same as the one specified in Algorithm 5. It can be further modified by using the training methods of double DDQN (Algorithm 6). There is no real recipe on what algorithm is going to work better than another. Still, in the literature, it has been shown that in runs with multiple million episodes, dueling DQN and double DQN tend to outperform in performance the classic DQN algorithm.

3.3 Examples

In this section, we implement the DQN agent and the variants mentioned in the last section using the Gym, TensorFlow 2, and Keras-rl2 libraries for different MDPs.

3.3.1 Optimal control

Acrobot

We begin with a simple optimal control problem known as Acrobot, where a double pendulum is controlled by rotating the central pivot as shown in Figure 3.2. However, this task is challenging to solve using table q-learning, since the state space (in theory) is infinite. It is described by two angles θ_1, θ_2 and two angular velocity $\dot{\theta_1}, \dot{\theta_2}$. On top of this, the double pendulum is a chaotic mechanical system. If we let the actions be

discrete, where the applied torque can be positive, negative, or zero, this task becomes suitable for the DQN algorithm.

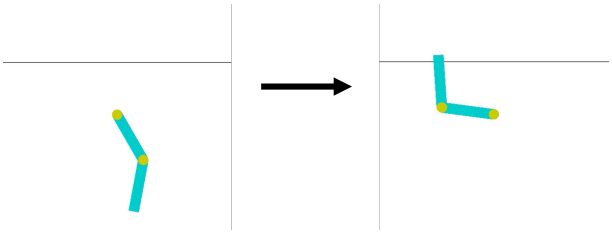


FIGURE 3.2: Acrobot environment. The task of this MDP is to swing the double pendulum in such a way that it crosses the black line.

We choose to approximate the agent with the neural architecture shown in Figure 3.3, composed of two 32-neurons hidden layers. We use an input state of 6 dimension defined as

$$s = \begin{pmatrix} \cos(\theta_1) \\ \sin(\theta_1) \\ \cos(\theta_2) \\ \sin(\theta_2) \\ \dot{\theta_1} \\ \dot{\theta_2} \end{pmatrix}$$

and output three possible actions corresponding to applying a $+1$, -1, or 0 torque to the central pivot. The reward used for this MDP is always -1 unless the double pendulum crosses the black line. In the latter (terminal) state, the reward is 0.

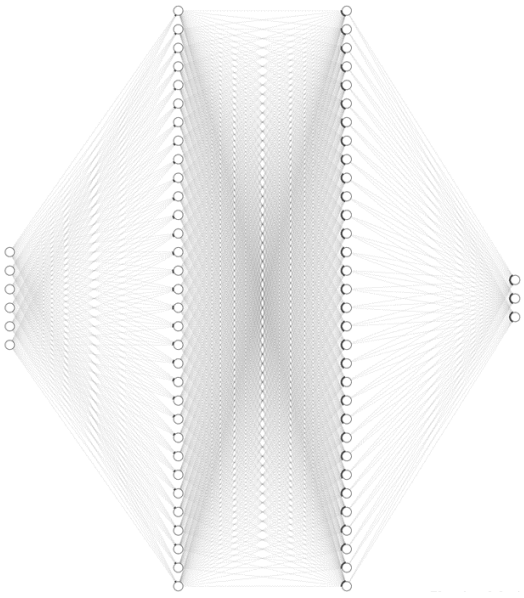


FIGURE 3.3: Model used to approximate the Q function of the "Acrobat-v1" environment.

After training the agent for $100,000$ episodes using the Adam optimizer with a learning rate of 10^{-3} and a mean absolute error loss function, the agent takes, on average, roughly 150 steps to solve the task as seen in the learning curve of Figure 3.4. It is important to note that we see multiple spikes with very low rewards achieved by the DQN agent throughout the training. The agent's exploration causes this behavior, and for unexplored states, which are the most likely to encounter in a continuous

setting MDP, the approximation of the optimal Q function could be very far from the actual value, causing the observed spikes.

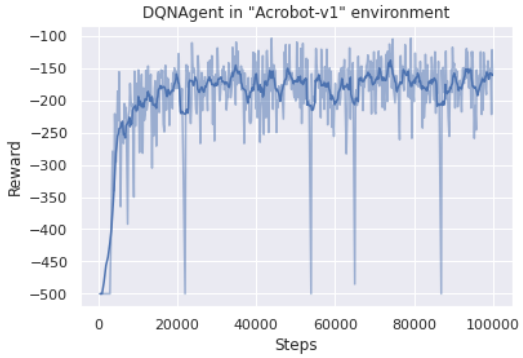


FIGURE 3.4: Smoothed learning curve of a DQN agent with two 32-node hidden layers with ReLu activation functions and a Boltzmann policy smoothed out for 10 episodes.

3.3.2 Atari Games

We now focus our attention on more complex MDPs, namely, Atari games, whose states are composed of an image, an array of intensities for every pixel of the image. In this case, the state space is $\mathcal{S} \cong [0,1]^{210\times160\times3}$ and the action space are the Atari playing buttons, which consist of 6 elements: 'NOOP', 'FIRE', 'RIGHT', 'LEFT', 'RIGHTFIRE', 'LEFTFIRE'. If we decided to use the image array as the input of some convolutional neural network, we would be using a lot of redundant information. A way to reduce redundancy and hence improve efficiency for training is by pre-processing the states to reduce the dimensionality, therefore, reducing the number of weights of the network. We can convert a colored 210×160 image into an 84×84 monochromatic image, reducing the state space dimension by a factor of 93%. This pre-processing keeps the essential correlations of the image while drastically reducing the input space. After doing this procedure, we process the new image with various convolutional layers. Namely, for all Atari games, we used the same neural network architecture shown in Figure 3.5.

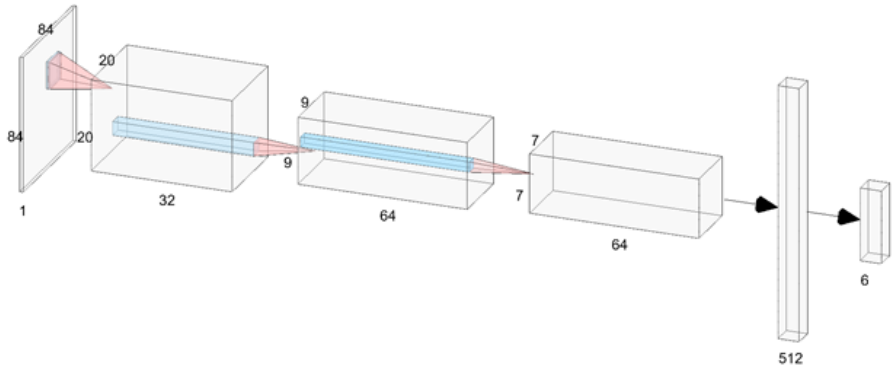


FIGURE 3.5: Model used to approximate the Q function of the Atari games.

Additionally, it is convenient to use as input not one but multiple images in different time steps for the prediction of actions so that the DQN agent captures a notion

of motion. The number of images used to predict an action is known as a *window*, and for the following examples, it is taken to be 4.

Pong

Beginning with the simplest Atari game, pong is a game where an agent controls a paddle and plays against the computer. This is a stochastic environment since the action of the computer is probabilistic, and since the transition kernel is unknown (unless we take a look at the code of the game), the reasonable way of solving this MDP is via a model-free reinforcement learning algorithm such as DQN. The winner side is that who scores the most goals while blocking the ball from passing the paddle. We start by doing a pre-processing of the state image as shown in Figure 3.6. It is clear from this picture that the qualitative aspects of the game are preserved. In this case, we can quickly identify the paddles and the ball after the image processing.

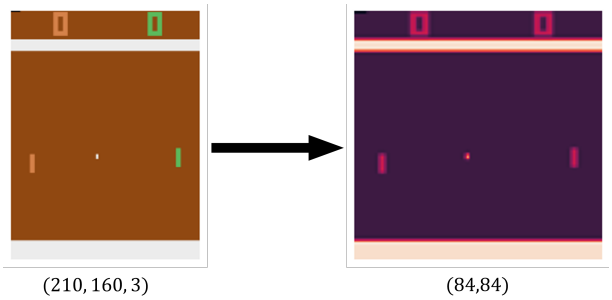


FIGURE 3.6: Processing environment observation to reduce dimensionality.

In this game, the reward depends on the number of goals made and received by the opponent, particularly their difference. We run the three DQN agents discussed in this chapter for the pong environment and obtain the learning curves from Figure 3.7. We successfully train the agents to win against the computer, which is shown in the positive average reward of the agents.

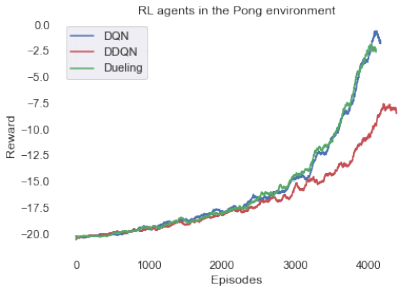


FIGURE 3.7: Smoothed reward (average over 100) of different DQN variants in the pong environment.

Performance-wise, for the number of episodes that the agent was trained (equivalent to 10 million steps), we see that the worst-performing agent is the double DQN. Whereas both DQN and dueling DQN perform similarly. We can furthermore see this decreased expected reward in the sampled mean value of $\max Q(s, a)$ for some number of states s (Figure 3.8).

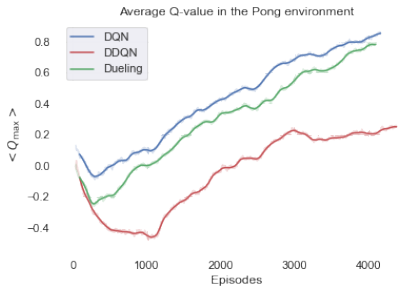


FIGURE 3.8: Smoothed mean $\max Q(s, a)$ (average over 100) of different DQN variants in the pong environment.

Space Invaders

We now switch our focus to a more complex game called Space Invaders. The game consists of a spaceship that needs to shoot and destroy alien spaceships. In addition, it needs to avoid enemy fire and prevent the alien spaceships from reaching the ground. This game is much harder to beat than pong is since the relevant action space is larger, and there are more stochastic events, such as the firing of different spaceships.

As we did before, we pre-process the input image as shown in Figure 3.9. It is essential to check that the pre-processed image does not remove any critical element of the game. Sometimes, depending on how the state space is reduced, the pixels of some color might be ignored. This is a problem since we could be losing key information about the state. In this case, we could lose information about the location of a spaceship or an enemy bullet that could harm the player's ship.

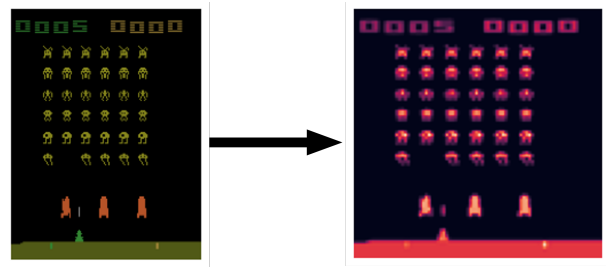


FIGURE 3.9: Processing environment observation to reduce dimensionality of the states.

We again train three different agents using DQN, DDQN, and dueling DQN. The three agents were trained for a total of 10 million steps, using an Adam optimizer with a learning rate of 10^{-4} and a replay buffer with a memory limit of $1'500,000$ states. We see a similar behavior of the agents' learning curves in this environment (Figure 3.10) compared to the pong environment, with the notable difference that for space invaders, the three algorithms have a comparable performance.

The similarity in the expected reward can be viewed on the estimated $\max Q(s, a)$ function as shown in Figure 3.11. An important point to mention is that the order between the estimated average $Q_{\max}$ value is the same as the pong game. Thus, while the DQN agent tends to overestimate the value function, the DDQN notably reduces this estimate.

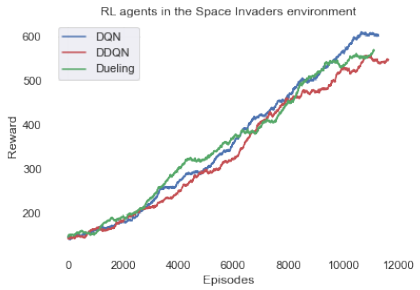


FIGURE 3.10: Smoothed reward (average over 500) of different DQN variants in the space invaders environment.

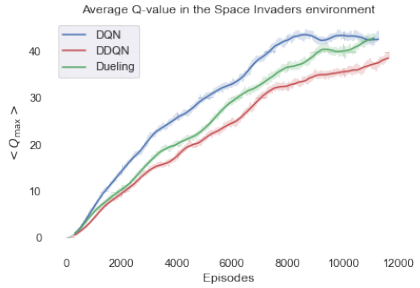


FIGURE 3.11: Smoothed mean $\max Q(s, a)$ (average over 100) of different DQN variants in the pong environment.

Breakout

Our last case of study is the Breakout environment, in which a paddle is moved to bounce a ball back and forth, and the goal is to destroy the bricks located at the top. Each brick and the direction of impact with the paddle will give a different speed to the ball. This game, in theory, is deterministic but chaotic due to its high-dimensional state space. Thus minor variations from the beginning of the game will strongly influence the states at a later time.

During the training, we pre-process, same as in the previous examples, every state as shown in Figure 3.12, and note that there is no information loss aside from the color of each brick. Since the height of each brick determines the color, there is no significant information loss, but if the color of the bricks were random, we would need another way of pre-processing these states.

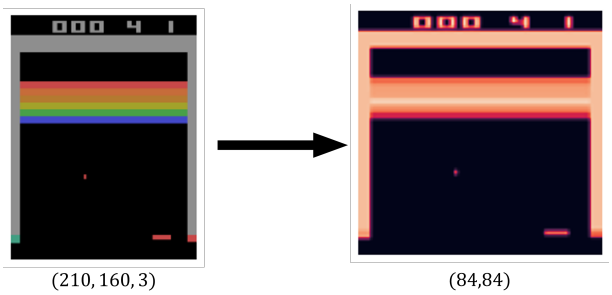


FIGURE 3.12: Processing environment observation to reduce dimensionality.

We train the three agents with the same hyper parameters as the previous examples and get successful learning curves shown in Figure 3.13.

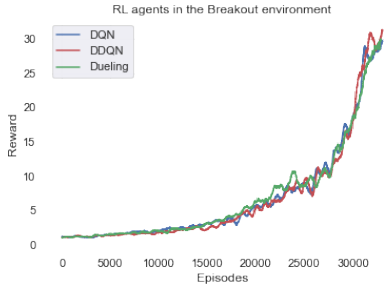


FIGURE 3.13: Smoothed reward (average over 500) of different DQN variants in the breakout environment.

For the mean Q value, we obtain the graph shown in Figure 3.14. This shows the difference in the value estimate for the three agents, where DDQN has a lower estimate for $\langle Q_{\max} \rangle$.

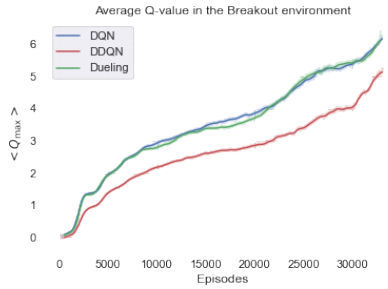


FIGURE 3.14: Smoothed mean $\max Q(s, a)$ (average over 100) of different DQN variants in the pong environment.

But what would happen if we did not stop here? What would happen if we let the agent training for more steps?

3.3.3 Should we keep training an agent?

For this subsection, we decided to keep training the agent of the game Breakout. So we trained it for another 5 million steps and found an exciting but still expected result. In this second round of training, the DQN agent discovered a trick to get higher rewards, as shown in Figure 3.15. The maneuver consists of breaking a hole on the side of the map to get the ball on the top of the bricks, as shown in Figure 3.16. The trick made the agent earn many more rewards than the policy followed in previous training stages.

A problem that must be pointed out is the non-increase of the average reward after the DQN agent discovered this trick. This can be explained since the replay buffer has a finite size and the DQN agent has an epsilon greedy behavior policy. Since getting the ball on top of the map requires precision, the probability of happening will be proportional to $(1 - \epsilon)^n$ where n is the number of the actions needed to do this task. This will lead to an average reward that does not increase. If we want to see an improvement, we must further fine-tune our DQN agent by further decreasing this ϵ exploratory constant, and increasing the size of the replay buffer will also lead to more accurate tuning.

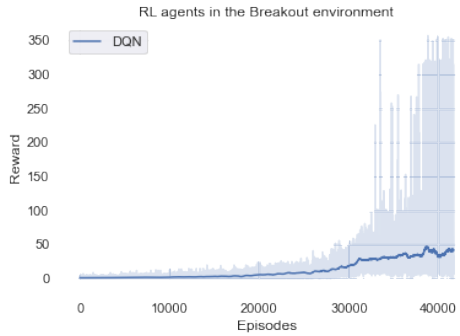


FIGURE 3.15: Reward of different DQN variants in the breakout environment.

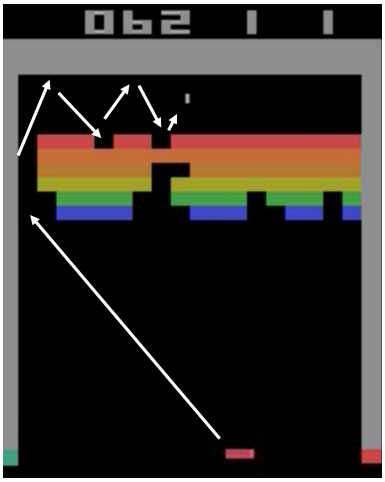
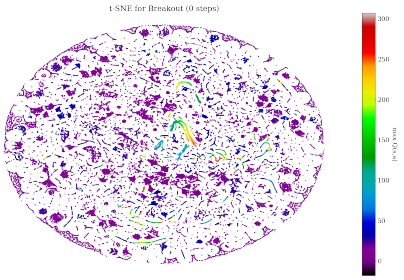


FIGURE 3.16: Policy found after 10M steps for the Breakout game. It exploits the topology of the bricks by opening a channel on the side and passing the ball to the upper side of the bricks.

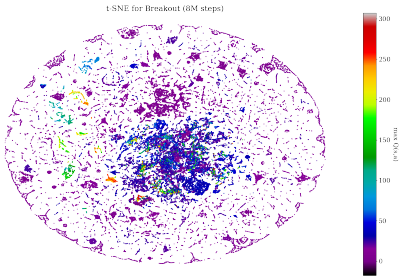
We can qualitatively see the learning of the DQN agent using a t-SNE plot[2] of the feature vectors of the last hidden layer for several input states of the DQN agent at different stages of learning. We did this for three stages of the total training: 0 steps, 8 million steps, and 15 million steps. The tSNE plots are shown in Figure 3.17. These plots were generated by letting the DQN agent (with 15 million training steps) play the breakout game for several hours and saving every state and each feature vector of the last hidden layer with its corresponding predicted value function. These multiple 512 dimensional vectors were then mapped to two-dimensional plots using the t-SNE algorithm—this map clusters nearby vectors. Thus, an agent that has learned to differentiate between "good" and "bad" states will tend to represent "good" (and "bad") states with feature vectors of similar components. The agent's ability to accurately represent the state will then be reflected by seeing states of a similar value function in the same cluster. This behavior is something that can be clearly seen in Figure 3.17, where at the beginning of training, states are located randomly in the 512 feature space, and at the end of the training, states will be located according to their future reward.

For this simple environment and orders of magnitude of training, we see that increasing the training steps leads to better network performance. Despite this, this

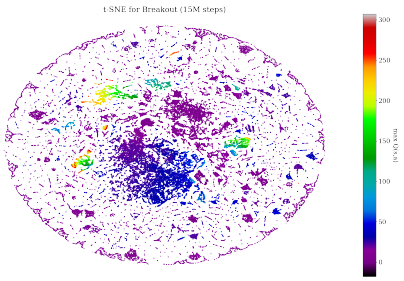
[2]The t-SNE embedding is explained in Appendix A.



(A) 0 steps



(B) 8 million steps



(E) 15 million steps

FIGURE 3.17: The tSNE plot for the state representation at the last hidden layer with 512 neurons after different steps of training.

is not always the case for every environment. For example, sometimes, the agent could over-fit the sampled data, leading to a lower average reward. Therefore, it is always necessary to track the average rewards and decide when is the correct time to stop.

Chapter 4

Conclusions and outlook

We have studied and implemented a small class of algorithms known as deep Q-learning, which are inspired by the mixture between reinforcement learning and deep learning. The main goal of these algorithms is to solve a discounted Markov decision problem in a high-dimensional state space, where classical reinforcement learning algorithms cannot succeed. The deep neural networks are used to approximate, in this case, the Q-value, also known as the quality, of a state-action pair.

In the first chapter, we have introduced to the reader the basic ideas of dynamic programming and reinforcement learning and some examples of MDPs that are solved using planning and model-free algorithms. Then, we introduce neural networks in the second chapter. These are function approximators that must be tuned to an optimal fit. This tuning depends on multiple hyperparameters, and choosing the optimal ones is not a trivial task. This chapter talks about the different optimizers, loss functions, and architectures that can be chosen to fit a given data set. Finally, in the third chapter, we mix these two concepts together and mention how researchers have dealt with the main problems that arise when merging these two areas, such as training a neural network with correlated data.

Using the aid of multiple libraries, such as Gym, Tensorflow 2.0, and Keras-rl2, we implemented numerous DQN agents. We began by training a simple DQN agent on an optimal control problem called Acrobot, where the Q function is approximated using a neural network with two 32-node hidden layers. After this, we implemented three variants of the DQN agent (DQN, double DQN (DDQN), and dueling DQN) for three different Atari environments: Pong, SpaceInvaders, and Breakout. These nine different agents were trained for 10 million steps using the same architecture for the DQN agent. We analyzed and compared the average reward and the average $Q_{\max}$ state-action function throughout the training process of these nine agents. We did not see a significant difference in the average reward between the three DQN variants for the number of training steps chosen. Still, we expect to have a notable difference for more training steps as mentioned in the literature. Furthermore, we saw the expected difference in the predicted $\langle Q_{\max} \rangle$ state-action function between the DDQN agent and the other two agents. Finally, we decided to take one DQN agent further, the DQN agent for the Atari breakout game and trained it for another 5 million episodes. After doing this, the agent's weights changed from a local minimum to a lower local minimum, where now the network was able to find a trick that consisted in break one side of the bricks and sending the ball to the top part, giving the agent a much higher reward than before. In addition, the feature vectors of the last hidden layer of several states were plotted using a tSNE map on different training stages of the training to visualize the representation of multiple states throughout the learning process.

Deep reinforcement learning is a tool that has shown incredible success in many fields throughout the past few years. This thesis showed how this simple idea could solve complex games, which would be near-impossible to solve via previously proposed

methods due to the massive computational overhead. But this is just a tiny grain of sand of a vast ocean full of deep reinforcement learning applications to solve problems our community faces that can be formalized as MDPs, from robotic control for surgery to autonomous driving for commuting. Thus, deep Q-learning takes us one step closer to artificial general intelligence, and on its way, it brings human society new technology for increasing its life quality and expectancy.

Appendix A

t-SNE

The t-Distributed Stochastic Neighbor Embedding [29], also known as t-SNE, is a technique that allows visualizing high-dimensional data $\{x_1, \ldots, x_N\}$ inspired in a previous algorithm called Stochastic Neighbor Embedding [30]. The main idea behind this algorithm is to use some similarity metric between two points x_i and x_j in the high-dimensional space to map them to two low-dimensional points y_i and y_j with a similar similarity. For this method, the similarity between two points $\{x_i, x_j\}$ is defined by the probability of choosing x_j as a neighbour of x_i given that this probability is proportional to a Gaussian distribution centered at x_i. This means that the conditional probability of choosing x_j given x_i is

$$p_{j|i} := P(x_j \mid x_i) = \frac{\exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma_i^2}\right)}{\sum_{k \neq i} \exp\left(-\frac{\|x_k - x_i\|^2}{2\sigma_i^2}\right)},$$

where σ_i^2 is the variance of the Gaussian distribution centered at x_i. If we define a cost function using this conditional probability we will encounter some trouble for outlier points, which a problem of the SNE algorithm [30]. The t-SNE deals with this by considering the symmetrized conditional probability

$$p_{ij} = \frac{p_{i|j} + p_{j|i}}{2N}.$$

To map these high-dimensional points to a set of low-dimensional points $\{y_1, \ldots, y_N\}$, we must first consider some similarity measure between y_i and y_j. Two common ways of defining the similarity between y_i and y_j is with

$$q_{ij} = \frac{\exp\left(-\|y_i - y_j\|^2\right)}{\sum_{k \neq l} \exp\left(-\|y_k - y_l\|^2\right)},$$

and with

$$q_{ij} = \frac{\left(1 + \|y_i - y_j\|^2\right)^{-1}}{\sum_k \sum_{k \neq l} \left(1 + \|y_k - y_l\|^2\right)^{-1}}.$$

Finally, to find such points $\{y_i\}_{i=1}^N$ with a similar probability distribution q_{ij} to the one of the original data set p_{ij} we perform a gradient-based optimization of the Kullback-Leiber divergence

$$f(y_1, \ldots, y_N) = KL(P||Q) = \sum_{ij} p_{ij} \log \frac{p_{ij}}{q_{ij}}.$$

In the t-SNE case (with symmetric p_{ij}), the gradients of f are given by

$$\frac{\partial f}{\partial y_i} = 4 \sum_j (p_{ij} - q_{ij}) (y_i - y_j)$$

for the former choice of q_{ij}, and

$$\frac{\partial f}{\partial y_i} = 4 \sum_j (p_{ij} - q_{ij}) (y_i - y_j) \left(1 + \|y_i - y_j\|^2\right)^{-1}$$

for the latter. Finally, there is a hyper-parameter that we must choose to do this algorithm, which is the variances σ_i^2 of the Gaussian distributions. The way these variances are chosen are such that the perplexity, defined by

$$PP(P_i) = 2^{H(P_i)} = 2^{-\sum_k p_{k|i} \log p_{k|i}},$$

is equal to some predefined value, which is generally a number smaller than 100. It is clear from this definition that the larger the perplexity, the larger the variances σ_i^2 are.

Bibliography

[1] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*, MIT press, 2018.

[2] S. Almahdi and S. Y. Yang, Expert Systems with Applications **87**, 267 (2017).

[3] K.-T. Song and W.-Y. Sun, Journal of Intelligent and Robotic Systems **21**, 221 (1998).

[4] M. Y. Niu, S. Boixo, V. N. Smelyanskiy, and H. Neven, npj Quantum Information **5**, 33 (2019).

[5] C. J. Watkins and P. Dayan, Machine learning **8**, 279 (1992).

[6] G. Cybenko, Mathematics of control, signals and systems **2**, 303 (1989).

[7] V. Mnih et al., arXiv preprint arXiv:1312.5602 (2013).

[8] J. Schrittwieser et al., Nature **588**, 604 (2020).

[9] D. Silver et al., arXiv:1712.01815 [cs] (2017), arXiv: 1712.01815.

[10] J. Fan, Z. Wang, Y. Xie, and Z. Yang, arXiv:1901.00137 [cs, math, stat] (2020), arXiv: 1901.00137.

[11] S. Gu, E. Holly, T. Lillicrap, and S. Levine, arXiv:1610.00633 [cs] (2016), arXiv: 1610.00633.

[12] A. Franceschetti, E. Tosello, N. Castaman, and S. Ghidoni, arXiv:2005.02632 [cs] (2020), arXiv: 2005.02632.

[13] T. Jaakkola, M. Jordan, and S. Singh, Convergence of Stochastic Iterative Dynamic Programming Algorithms, in *Advances in Neural Information Processing Systems*, volume 6, Morgan-Kaufmann, 1994.

[14] D. A. Roberts, S. Yaida, and B. Hanin, arXiv preprint arXiv:2106.10165 (2021).

[15] J. Carrasquilla and R. G. Melko, Nature Physics **13**, 431 (2017).

[16] S. Hochreiter and J. Schmidhuber, Neural computation **9**, 1735 (1997).

[17] A. Vaswani et al., Attention is all you need, in *Advances in neural information processing systems*, pages 5998–6008, 2017.

[18] T. B. Brown et al., arXiv preprint arXiv:2005.14165 (2020).

[19] M. Chen et al., arXiv preprint arXiv:2107.03374 (2021).

[20] A. Dosovitskiy et al., arXiv preprint arXiv:2010.11929 (2020).

[21] L. Chen et al., arXiv preprint arXiv:2106.01345 (2021).

[22] H. Kushner and G. G. Yin, *Stochastic approximation and recursive algorithms and applications*, volume 35, Springer Science & Business Media, 2003.

[23] Y. Nesterov, A method for unconstrained convex minimization problem with the rate of convergence $o(1/k^2)$, 1983.

[24] D. P. Kingma and J. Ba, arXiv preprint arXiv:1412.6980 (2014).

[25] M. Riedmiller, Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method, in *European conference on machine learning*, pages 317–328, Springer, 2005.

[26] J. Frankle and M. Carbin, arXiv preprint arXiv:1803.03635 (2018).

[27] H. Van Hasselt, A. Guez, and D. Silver, Deep reinforcement learning with double q-learning, in *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[28] Z. Wang et al., Dueling network architectures for deep reinforcement learning, in *International conference on machine learning*, pages 1995–2003, PMLR, 2016.

[29] L. Van der Maaten and G. Hinton, Journal of machine learning research **9** (2008).

[30] G. Hinton and S. T. Roweis, Stochastic neighbor embedding, in *NIPS*, volume 15, pages 833–840, Citeseer, 2002.

[31] M. L. Puterman, *Markov decision processes: discrete stochastic dynamic programming*, John Wiley & Sons, 2014.

[32] M. A. Nielsen, *Neural networks and deep learning*, volume 25, Determination press San Francisco, CA, 2015.

[33] J. R. Birge and F. Louveaux, *Introduction to Stochastic Programming*, Springer Series in Operations Research and Financial Engineering, Springer New York, New York, NY, 2011.

[34] A. Shapiro, D. Dentcheva, and A. P. Ruszczyński, *Lectures on stochastic programming: modeling and theory*, Number 9 in MPS-SIAM series on optimization, Society for Industrial and Applied Mathematics : Mathematical Programming Society, Philadelphia, 2009, OCLC: ocn402540716.

[35] D. Bertsekas and J. Tsitsiklis, Neuro-dynamic programming: an overview, in *Proceedings of 1995 34th IEEE Conference on Decision and Control*, volume 1, pages 560–564 vol.1, 1995, ISSN: 0191-2216.

[36] A. LeNail, Journal of Open Source Software **4**, 747 (2019).